Big Data compendium

Jacob Clements

March 2024

Comments

In this compendium I have one section for each of the lectures given by the professor. The information is mainly based on the lectures given, and supplemented (if necessary) by the course curriculum papers. Usually there will be subsections for specific topics within the lecture and if there are main topics (larger width) these will have subsubsections. For the discussion topics, note that these are transcriptions from the class so might not be that reader friendly.

NB!: The lecturer has stated that he likes to give original exam questions, hence, must learn the curriculum properly, and learning previous exam questions will not necessarily help on the exam.

Introduction to Big Data

Terminology

We start by defining some key concepts which will be employed throughout the course:

Structured data: well-defined fields, i.e. represented in tables.

Unstructured data: Typical "by human for human", i.e. text messages, emails etc.

Semi-strucutred data: Self-decribing (with tags) like XML and JSON. This type of data can have variable semantic strictness. For instance, some might say birht day whilst others date of birth. But there is some annotation, less or more formal.

Batch-oritented: It is about processing. Running a sequence of computer programs with no human interaction (compared to interactive execution). Also says something about timeframes expected. It is typically computation for throughput not latency.

Batch-Processing: Batch processing is a method of data processing where a group of transactions is collected over a period of time and then processed all at once. This approach is particularly useful for processing large volumes of data where real-time computation is not required. Batch processing systems are designed to be efficient and economical, executing data-intensive operations as a single unit, often during off-peak hours, to optimize resource use and minimize impact on performance.

Near-realtime (NRT): Small delay between when data is available and when it is processed.

Real-time: Guarenteed time for when data was available until it is processed.

Streaming data: "Ordered sequence of instances", like sensory data or twitter-messages. Typical with high frequency where processing occurs immediately without knowing the entire sequence (not even the ones previously arrived).

Big Data Defintions

"Big data is a broad term for data sets so large or complex that traditional data processing applications (i.e., DBMS) are inadequate for capturing/storing/managing/analyzing." — McKinsey

"Big data is high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making." — Gartner

The 5 Vs

These are good characteristics of big data.

Volume: The immense amount of data generated every second from social media, cell phones, cars, credit cards, M2M sensors, images, video, and more.

Velocity: The speed at which new data is generated and the speed at which data moves around.

Variety: Data comes in various formats: structured numeric data in traditional databases; unstructured text documents, emails, videos, audios, stock ticker data, and financial transactions.

Veracity: With many forms of big data, quality and accuracy are less controllable (e.g., sensor data is often noisy).

Value: This is about turning our data into value. It's the end-goal in utilizing big data.

Key techniques in Big Data

These are just a few of the techniques used in Big data:

- Massive parallelism
- Data storage
- Network
- Databases/Queries
- Supercomputing
- Data mining

- Machine learning
- Information retrieval
- Visualization

Why Big Data now?

- More data collected and stored
  - Web (including news): 100T URLs are known
  - Social media, user-generated data
    - Facebook: 4 million likes/minute, 1 billion stories/day.
    - Twitter: 500 million tweets per day (2022), that is 6000/second!
  - Transactional data (ecommerce, logistics, telecom)
  - Internet of Things (sensors, RFID, etc.)
- Open Source
  - GNU, Linux, Hadoop, Spark...
- Standard powerful software and hardware

Some Examples of Applications

- Recommendation systems
- Financial analysis
- Social Media Monotoring
- Traffic navigation
- Health (monotoring etc.)

General Big Data Architecture

We first show an image of how we imagine the architecture before supplementing it with an explanation. Figure:

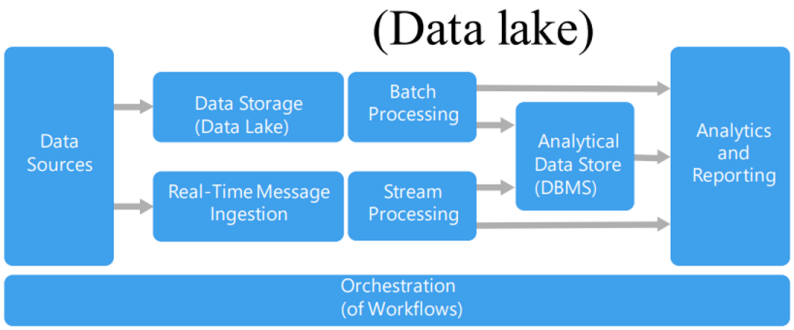


Figure 1: Visualization of Data lake

In the context of big data architectures, the concept of a Data Lake represents a centralized repository where all data from various sources is stored, irrespective of its volume or variety. This storage is analogous to a lake where data is accumulated and rarely updated, with new information continuously added.

The Data Lake is employed for massive batch processing, where extensive data sets are processed in chunks. Simultaneously, the architecture supports real-time message ingestion for immediate data flow from sources like sensors.

Stream processing works in parallel to handle real-time or near real-time data updates. The results from batch and stream processing often feed into a Database Management System (DBMS), which holds precomputed, essential data subsets for efficient query and analysis. These subsets represent the most significant results of the computations done on the full data set.

For the purpose of analytics and reporting, the data is further processed to produce interactive tools or scheduled reports that summarize insights, like average metrics from specific subsets of the data.

Additionally, an orchestration layer is crucial for managing workflows, scheduling jobs, and directing data flows efficiently within the system. This orchestration is vital for system-level organization and can benefit from standardization across different systems.

This framework is a generalized blueprint for data management, involving a vast repository for all data, concurrent stream processing, optimized data stores for analysis, and a robust workflow orchestration system.

Big Data: Infrastructure and Tooling

Hadoop, initiated by Doug Cutting at Yahoo in 2006, laid the foundation for open-source big data systems. Inspired by Google's MapReduce and Google File System papers, Cutting implemented these concepts into what is now known as the Hadoop Distributed File System (HDFS) and the resource management tool YARN (Yet Another Resource Negotiator).

Hadoop and its suite of tools under the Apache Foundation became pivotal in big data processing, offering a distributed file system, resource management, and processing capabilities. Notable among related projects is Apache Spark, created by Matei Zaharian, which introduced a more performant computing model compared to Hadoop's MapReduce.

These open-source projects are often distributed commercially with added proprietary tooling and support services by companies like Cloudera and Databricks. These distributions are made accessible on cloud platforms such as AWS, Azure, and Google Cloud, providing scalable big data infrastructure solutions without the need for local data center resources.

Quick Tour of Hadoop Before Lecture 2

Here is a slide of its ecosystem:

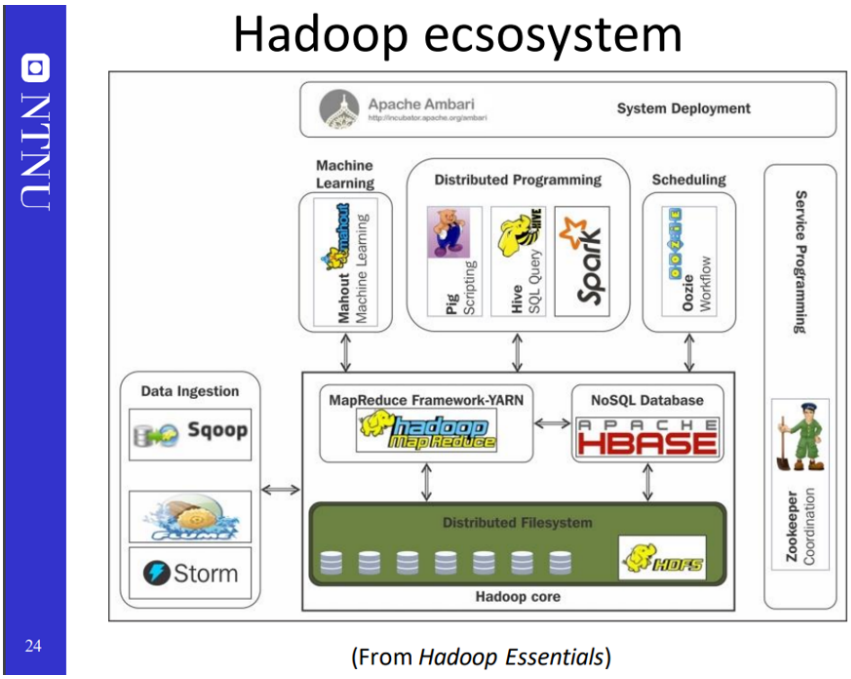


Figure 2: Hadoop ecosystem

The point of this slide is there is a bunch of systems in the Hadoop ecosystem. We will cover HDFS. We will cover MapReduce. We will cover and use Spark. Moreover, we will look into Storm which is a stream processing system.

Lecture 2: The Hadoop Distributed File System

For this section I steal most of the content from the curriculum paper. Before diving into the curriculum paper, we quickly state what it is good and bad for:

- Good:
  - (Very) large files
  - Stream-access: high throughput

  - Localized/affinitized process of data (reduce communication)
  - standard hardware
  - Fault-tolerant
- bad:
  - low-latency data access
  - Large number of small files
  - Multiple writer scenerios
  - Random updates (append only system)

Here is an illustration of hwo distributed file systems behave compared to traditional ones:

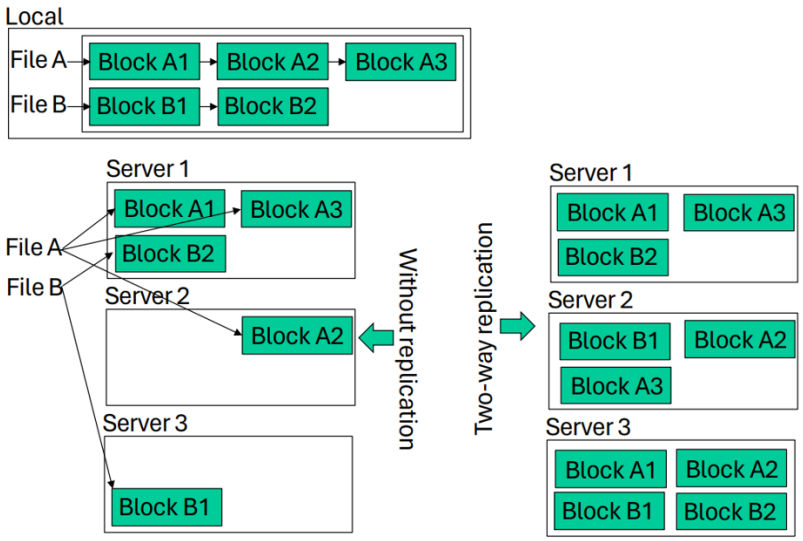


Figure 3: Hadoop ecosystem

Abstract

The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size.

Introduction and related work

Hadoop provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce paradigm. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application

computations in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and IO bandwidth by simply adding commodity servers.
HDFS is the file system component of Hadoop.
HDFS stores file system metadata and application data separately. HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols. The DataNodes in HDFS do not use data protection mechanisms such as RAID to make the data durable. Instead, the file content is replicated on multiple DataNodes for reliability. While ensuring data durability, this strategy has the added advantage that data transfer bandwidth is multiplied, and there are more opportunities for locating computation near the needed data.

Architecture

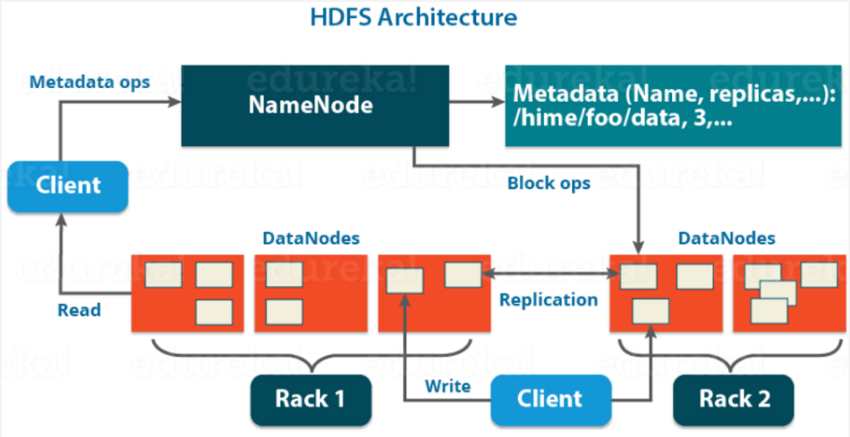


Figure 4: HDFS Architecture

NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file) and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes (the physical location of file data). An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the client. When writing data, the client requests the NameNode to nominate a suite of three DataNodes to host the block replicas. The client then writes data to the DataNodes in a pipeline fashion. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently.
HDFS keeps the entire namespace in RAM. The inode data and the list of blocks belonging to each

file comprise the metadata of the name system called the image. The persistent record of the image stored in the local host's native files system is called a checkpoint. The NameNode also stores the modification log of the image called the journal in the local host's native file system. For improved durability, redundant copies of the checkpoint and journal can be made at other servers. During restarts the NameNode restores the namespace by reading the namespace and replaying the journal. The locations of block replicas may change over time and are not part of the persistent checkpoint.

DataNodes

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.

During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either does not match that of the NameNode the DataNode automatically shuts down.

The namespace ID is assigned to the file system instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus preserving the integrity of the file system.

The consistency of software versions is important because incompatible version may cause data corruption or loss, and on large clusters of thousands of machines it is easy to overlook nodes that did not shut down properly prior to the software upgrade or were not available during the upgrade. A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID.

After the handshake the DataNode registers with the NameNode. DataNodes persistently store their unique storage IDs. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that.

A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block id, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-todate view of where block replicas are located on the cluster.

During normal operation DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes.

Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's space allocation and load balancing decisions.

The NameNode does not directly call DataNodes. It uses replies to heartbeats to send instructions

to the DataNodes. The instructions include commands to:

- replicate blocks to other nodes;
- remove local block replicas;
- re-register or to shut down the node;
- send an immediate block report.

These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations.

HDFS Client

User applications access the file system using the HDFS client, a code library that exports the HDFS file system interface.
Similar to most conventional file systems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in the namespace. The user application generally does not need to know that file system metadata and storage are on different servers, or that blocks have multiple replicas.
When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of DataNodes is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in the figure below.

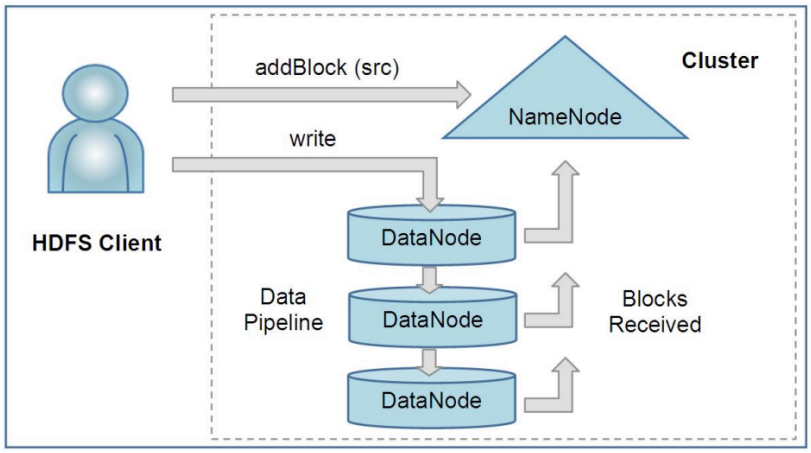


Figure 5: An HDFS client creates a new file by giving its path to the NameNode. For each block of the file, the NameNode returns a list of DataNodes to host its replicas. The client then pipelines data to the chosen DataNodes, which eventually confirm the creation of the block replicas to the NameNode.

Unlike conventional file systems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows an application to set the replication factor of a file. By default a file's replication factor is three. For critical files or files which are accessed very often, having a higher replication factor improves their tolerance against faults and increase their read bandwidth.

Upgrades, File System Snapshots

During software upgrades the possibility of corrupting the system due to software bugs or human mistakes increases. The purpose of creating snapshots in HDFS is to minimize potential damage to the data stored in the system during upgrades.
The snapshot mechanism lets administrators persistently save the current state of the file system, so that if the upgrade results in data loss or corruption it is possible to rollback the upgrade and return HDFS to the namespace and storage state as they were at the time of the snapshot.
The snapshot (only one can exist) is created at the cluster administrator's option whenever the system is started. If a snapshot is requested, the NameNode first reads the checkpoint and journal files and merges them in memory. Then it writes the new checkpoint and the empty journal to a new location, so that the old checkpoint and journal remain unchanged.
During handshake the NameNode instructs DataNodes whether to create a local snapshot. The local snapshot on the DataNode cannot be created by replicating the data files directories as this will require doubling the storage capacity of every DataNode on the cluster. Instead each DataNode creates a copy of the storage directory and hard links existing block files into it. When the DataNode removes a block it removes only the hard link, and block modifications during appends use the copy-on-write technique. Thus old block replicas remain untouched in their old directories. The cluster administrator can choose to roll back HDFS to the snapshot state when restarting the

system. The NameNode recovers the checkpoint saved when the snapshot was created. DataNodes restore the previously renamed directories and initiate a background process to delete block replicas created after the snapshot was made. Having chosen to roll back, there is no provision to roll forward. The cluster administrator can recover the storage occupied by the snapshot by commanding the system to abandon the snapshot, thus finalizing the software upgrade.
System evolution may lead to a change in the format of the NameNode's checkpoint and journal files, or in the data representation of block replica files on DataNodes. The layout version identifies the data representation formats, and is persistently stored in the NameNode's and the DataNodes' storage directories. During startup each node compares the layout version of the current software with the version stored in its storage directories and automatically converts data from older formats to the newer ones. The conversion requires the mandatory creation of a snapshot when the system restarts with the new software layout version.
HDFS does not separate layout versions for the NameNode and DataNodes because snapshot creation must be an allcluster effort rather than a node-selective event. If an upgraded NameNode due to a software bug purges its image then backing up only the namespace state still results in total data loss, as the NameNode will not recognize the blocks reported by DataNodes, and will order their deletion. Rolling back in this case will recover the metadata, but the data itself will be lost. A coordinated snapshot is required to avoid a cataclysmic destruction.

File I/O Operations and Replica Management

File Read and Write

An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.
The HDFS client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease by sending a heartbeat to the NameNode. When the file is closed, the lease is revoked. The lease duration is bound by a soft limit and a hard limit. Until the soft limit expires, the writer is certain of exclusive access to the file. If the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. If after the hard limit expires (one hour) and the client has failed to renew the lease, HDFS assumes that the client has quit and will automatically close the file on behalf of the writer, and recover the lease. The writer's lease does not prevent other clients from reading the file; a file may have many concurrent readers.
An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically 64 KB), the data are pushed to the pipeline. The next packet can be pushed to the pipeline before receiving the acknowledgement for the previous packets. The number of outstanding packets is limited by the outstanding packets window size of the client.
After data are written to an HDFS file, HDFS does not provide any guarantee that data are visible to a new reader until the file is closed. If a user application needs the visibility guarantee, it can explicitly call the hflush operation. Then the current packet is immediately pushed to the pipeline, and the hflush operation will wait until all DataNodes in the pipeline acknowledge the successful

transmission of the packet. All data written before the hflush operation are then certain to be visible to readers.

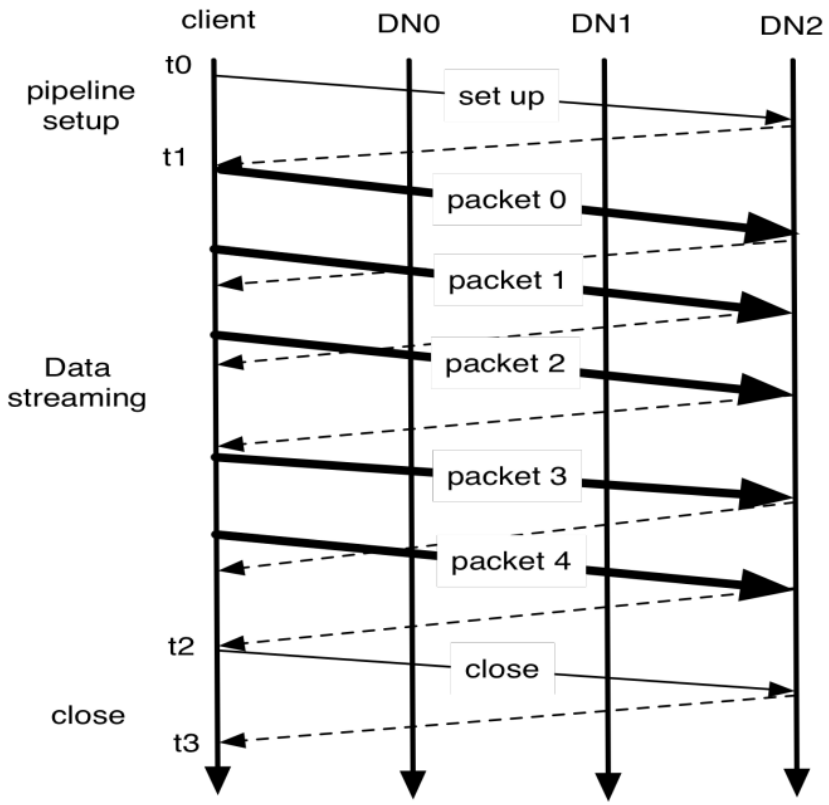


Figure 6: Data pipeline during block construction

If no error occurs, block construction goes through three stages as shown in the figure above illustrating a pipeline of three DataNodes (DN) and a block of five packets. In the picture, bold lines represent data packets, dashed lines represent acknowledgment messages, and thin lines represent control messages to setup and close the pipeline. Vertical lines represent activity at the client and the three DataNodes where time proceeds from top to bottom. From t0 to t1 is the pipeline setup stage. The interval t1 to t2 is the data streaming stage, where t1 is the time when the first data packet gets sent and t2 is the time that the acknowledgment to the last packet gets received. Here an hflush operation transmits the second packet. The hflush indication travels with the packet data and is not a separate operation. The final interval t2 to t3 is the pipeline close stage for this block. In a cluster of thousands of nodes, failures of a node (most commonly storage faults) are daily occurrences. A replica stored on a DataNode may become corrupted because of faults in memory, disk, or network. HDFS generates and stores checksums for each data block of an HDFS file. Checksums are verified by the HDFS client while reading to help detect any corruption caused

either by client, DataNodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. A DataNode stores checksums in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are shipped to the client. The client computes the checksum for the received data and verifies that the newly computed checksums matches the checksums it received. If not, the client notifies the NameNode of the corrupt replica and then fetches a different replica of the block from another DataNode

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

HDFS permits a client to read a file that is open for writing. When reading a file open for writing, the length of the last block still being written is unknown to the NameNode. In this case, the client asks one of the replicas for the latest length before starting to read its content.

The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes. However, many efforts have been put to improve its read/write response time in order to support applications like Scribe that provide real-time data streaming to HDFS, or HBase that provides random, realtime access to large tables.

Block Placement

For a large cluster, it may not be practical to connect all nodes in a flat topology. A common practice is to spread the nodes across multiple racks. Nodes of a rack share a switch, and rack switches are connected by one or more core switches. Communication between two nodes in different racks has to go through multiple switches. In most cases, network bandwidth between nodes in the same rack is greater than network bandwidth between nodes in different racks. The figure below describes a cluster with two racks, each of which contains three nodes.

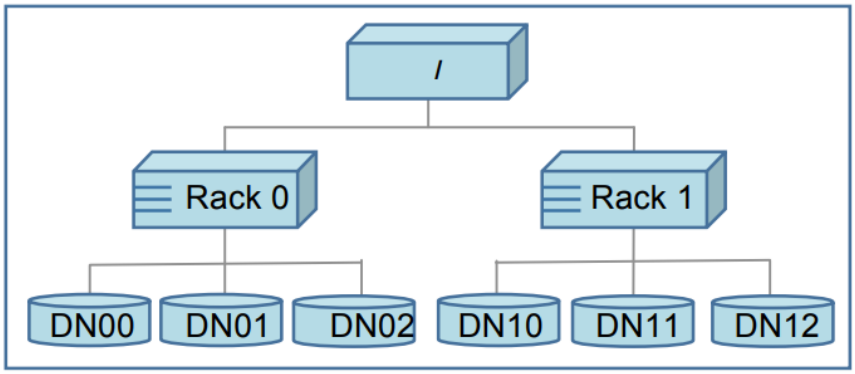


Figure 7: Cluster topology example

HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing up their distances to their closest common ancestor. A shorter distance between two nodes means that the greater bandwidth they can utilize to transfer data.
HDFS allows an administrator to configure a script that returns a node's rack identification given a node's address. The NameNode is the central place that resolves the rack location of each DataNode. When a DataNode registers with the NameNode, the NameNode runs a configured script to decide which rack the node belongs to. If no such a script is configured, the NameNode assumes that all the nodes belong to a default single rack.
The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. Currently HDFS provides a configurable block placement policy interface so that the users and researchers can experiment and test any policy that's optimal for their applications.
The default HDFS block placement policy provides a tradeoff between minimizing the write cost, and maximizing data reliability, availability and aggregate read bandwidth. When a new block is created, HDFS places the first replica on the node where the writer is located, the second and the third replicas on two different nodes in a different rack, and the rest are placed on random nodes with restrictions that no more than one replica is placed at one node and no more than two replicas are placed in the same rack when the number of replicas is less than twice the number of racks. The choice to place the second and third replicas on a different rack better distributes the block replicas for a single file across the cluster. If the first two replicas were placed on the same rack, for any file, two-thirds of its block replicas would be on the same rack.
After all target nodes are selected, nodes are organized as a pipeline in the order of their proximity to the first replica. Data are pushed to nodes in this order. For reading, the NameNode first checks if the client's host is located in the cluster. If yes, block locations are returned to the client in the order of its closeness to the reader. The block is read from DataNodes in this preference order. (It is usual for MapReduce applications to run on cluster nodes, but as long as a host can connect to the NameNode and DataNodes, it can execute the HDFS client.)
This policy reduces the inter-rack and inter-node write traffic and generally improves write performance. Because the chance of a rack failure is far less than that of a node failure, this policy does not impact data reliability and availability guarantees. In the usual case of three replicas, it can reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three.
The default HDFS replica placement policy can be summarized as follows:

- No Datanode contains more than one replica of any block.
- No rack contains more than two replicas of the same block, provided there are sufficient racks on the cluster.

Replication Management

The NameNode endeavors to ensure that each block always has the intended number of replicas. The NameNode detects that a block has become under- or over-replicated when a block report from a DataNode arrives. When a block becomes over replicated, the NameNode chooses a replica to remove. The NameNode will prefer not to reduce the number of racks that host replicas, and secondly prefer to remove a replica from the DataNode with the least amount of available disk

space. The goal is to balance storage utilization across DataNodes without reducing the block's availability.
When a block becomes under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. A background thread periodically scans the head of the replication queue to decide where to place new replicas. Block replication follows a similar policy as that of the new block placement. If the number of existing replicas is one, HDFS places the next replica on a different rack. In case that the block has two existing replicas, if the two existing replicas are on the same rack, the third replica is placed on a different rack; otherwise, the third replica is placed on a different node in the same rack as an existing replica. Here the goal is to reduce the cost of creating new replicas.
The NameNode also makes sure that not all replicas of a block are located on one rack. If the NameNode detects that a block's replicas end up at one rack, the NameNode treats the block as under-replicated and replicates the block to a different rack using the same block placement policy described above. After the NameNode receives the notification that the replica is created, the block becomes over-replicated. The NameNode then will decides to remove an old replica because the overreplication policy prefers not to reduce the number of racks.

Balancer

HDFS block placement strategy does not take into account DataNode disk space utilization. This is to avoid placing new—more likely to be referenced—data at a small subset of the DataNodes. Therefore data might not always be placed uniformly across DataNodes. Imbalance also occurs when new nodes are added to the cluster.
The balancer is a tool that balances disk space usage on an HDFS cluster. It takes a threshold value as an input parameter, which is a fraction in the range of (0, 1). A cluster is balanced if for each DataNode, the utilization of the node (ratio of used space at the node to total capacity of the node) differs from the utilization of the whole cluster (ratio of used space in the cluster to total capacity of the cluster) by no more than the threshold value.
The tool is deployed as an application program that can be run by the cluster administrator. It iteratively moves replicas from DataNodes with higher utilization to DataNodes with lower utilization. One key requirement for the balancer is to maintain data availability. When choosing a replica to move and deciding its destination, the balancer guarantees that the decision does not reduce either the number of replicas or the number of racks.
The balancer optimizes the balancing process by minimizing the inter-rack data copying. If the balancer decides that a replica A needs to be moved to a different rack and the destination rack happens to have a replica B of the same block, the data will be copied from replica B instead of replica A.
A second configuration parameter limits the bandwidth consumed by rebalancing operations. The higher the allowed bandwidth, the faster a cluster can reach the balanced state, but with greater competition with application processes.

Block Scanner

Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data. In each scan period, the block scanner adjusts the read bandwidth in order to complete the verification in a configurable period. If a client reads a complete

block and checksum verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica.
The verification time of each block is stored in a human readable log file. At any time there are up to two files in toplevel DataNode directory, current and prev logs. New verification times are appended to current file. Correspondingly each DataNode has an in-memory scanning list ordered by the replica's verification time.
Whenever a read client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt, but does not schedule deletion of the replica immediately. Instead, it starts to replicate a good copy of the block. Only when the good replica count reaches the replication factor of the block the corrupt replica is scheduled to be removed. This policy aims to preserve data as long as possible. So even if all replicas of a block are corrupt, the policy allows the user to retrieve its data from the corrupt replicas.

Decommisioning

The cluster administrator specifies which nodes can join the cluster by listing the host addresses of nodes that are permitted to register and the host addresses of nodes that are not permitted to register. The administrator can command the system to re-evaluate these include and exclude lists. A present member of the cluster that becomes excluded is marked for decommissioning. Once a DataNode is marked as decommissioning, it will not be selected as the target of replica placement, but it will continue to serve read requests. The NameNode starts to schedule replication of its blocks to other DataNodes. Once the NameNode detects that all blocks on the decommissioning DataNode are replicated, the node enters the decommissioned state. Then it can be safely removed from the cluster without jeopardizing any data availability.

Inter-Cluster Data Copy

When working with large datasets, copying data into and out of a HDFS cluster is daunting. HDFS provides a tool called DistCp for large inter/intra-cluster parallel copying. It is a MapReduce job; each of the map tasks copies a portion of the source data into the destination file system. The MapReduce framework automatically handles parallel task scheduling, error detection and recovery.

Lecture 3: MapReduce and Yarn

MapReduce

MapReduce is a programming framework for parallell and scalable processing of (very) large datasets. Its goals are listed as follows:

- process large files/data sets (many PBs)
- Many nodes (many thousands)
- Linear scalability (10x capacity

  $\rightarrow$

  10x performance)
- Hide complexity in deployment, parallelization and fault tolerance for application developers

- Distinguish between code for distributed execution and application code → change applications/algorithm easy
- Off-the-shelf machines

How it works

MapReduce is a processing technique that structures big data tasks into two main phases: Map and Reduce. In the Map phase, the input data is organized into key-value pairs and transformed into an intermediate state that suits the computation needs. This involves taking each element from the input, applying a mapping function, and emitting a list of new key-value pairs based on this function.

The Reduce phase then takes over, where all elements with the same key, generated by the Map phase, are aggregated. The reducer function is called for each key and processes the corresponding group of intermediate data, outputting the final result. The framework itself ensures that all keys are handled.

In essence, MapReduce works by:

1. Mapping: Converting input data into a new set of key-value pairs
2. Reducing: Combining all intermediate values associated with the same key to produce the final output.

Data from the Map phase is sharded by key so that all tuples with the identical key are assigned to the same reducer. This enables efficient processing as the data is sorted by key, streamlining the Reduce phase. This process efficiently handles vast datasets by distributing and parallelizing the job across multiple nodes, each performing a part of the task.

This scalable framework is critical for big data processing, enabling the handling of large-scale data across distributed systems.

MapReduce Examples: Count Words and Weather Data

We show pictures of the example of MapReduce:

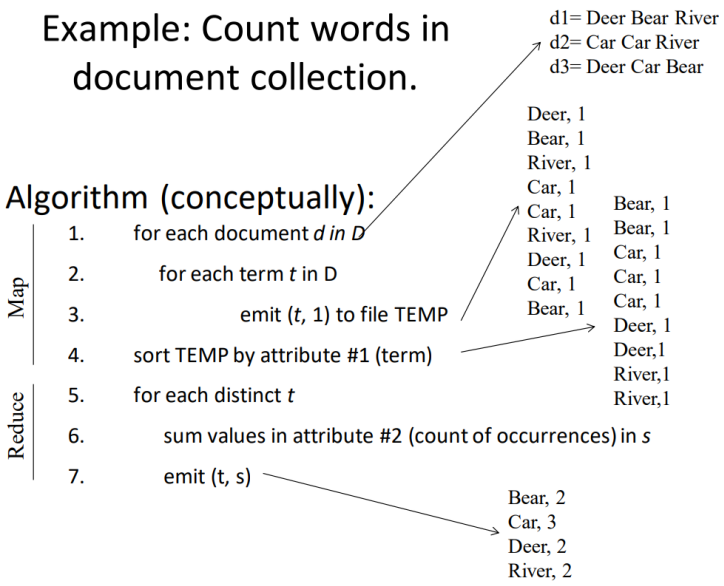


Figure 8: Step 1.

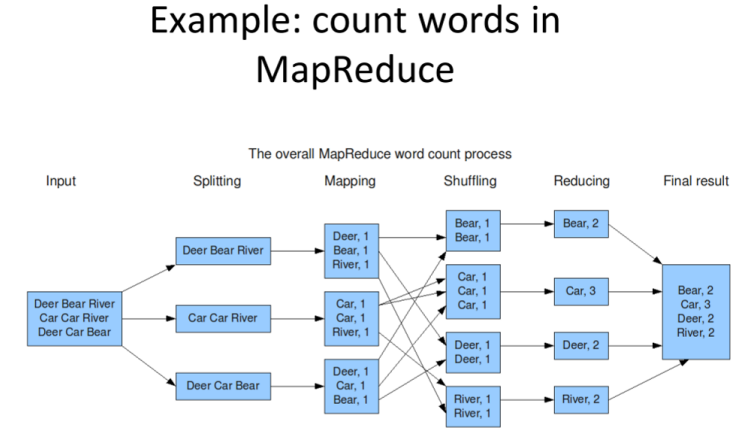


Figure 9: Step 2.

MapReduce version (high-level)

```
mapper (position, line-contents):
        for each word in line-contents:
                output (word, 1)

reducer (word, values):
        sum = 0
        for each value in values:
                sum = sum + value
        output(word, sum)
```

Figure 10: Step 3.

Example: count words in MapReduce

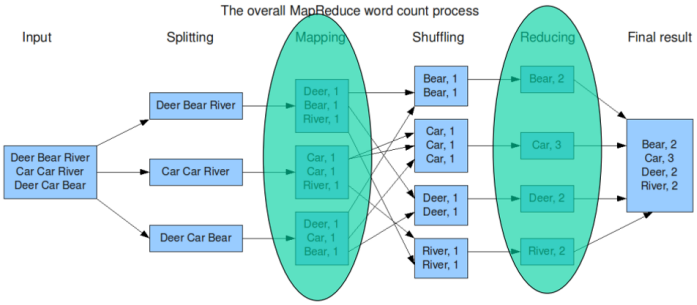


Figure 11: Step 4.

Here is our second example of MapReduce:

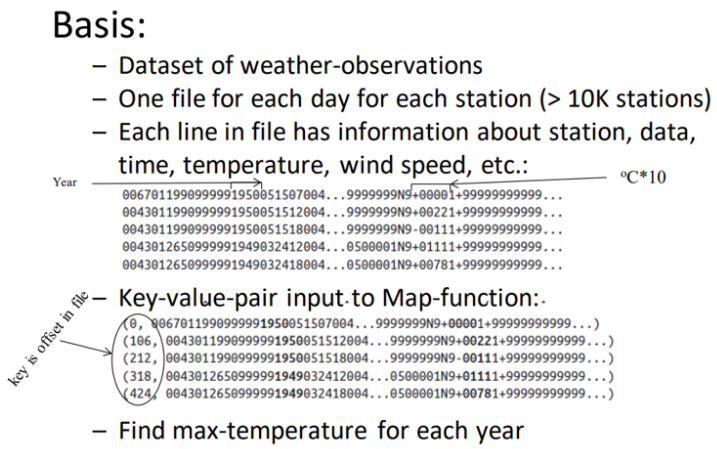


Figure 12: Step 1.

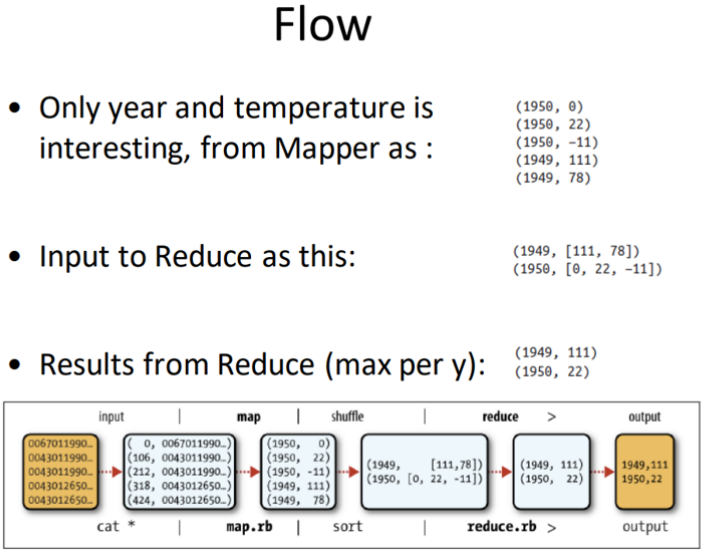


Figure 13: Step 2.

MapReduce Dataflow

Here we take a birds eyes view.

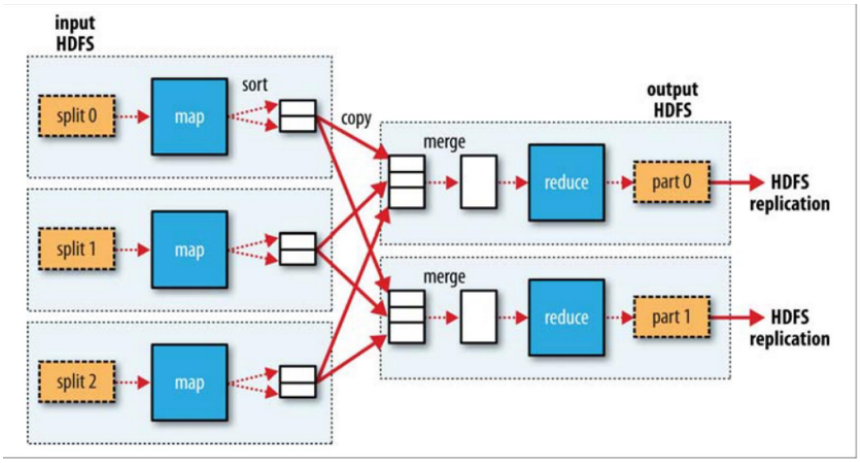


Figure 14: MapReduce dataflow with multiple reduce tasks

In the previous examples, we obviously looked at local files. Local file systems. Small files. However, connect this with what we talked about on HDFS, so data is partitioned on different nodes. Then the namenode, in HDFS, know where each of the different partitions are. Now this is knowledge that MapReduce can leverage. Because what you really want to do is to run mappers and reducers, on the same machines where your data is. We can now create the MapReduce jobs that is actually capable of running on all that data. And inherently distributed and processed close to the data. So MapReduce is really amazingly useful once you combine it with HDFS. And then you'll see we can have the splitter and the mapper. They all run in parallel. On each of these machines that have a part of the data on HDFS. And the output files can be equally partitioned, such that we "kind of" get rid of these choke points. So everything doesn't need to be waiting for the same hard drive or the same machine. This thing is mostly liberated from those constraints. That being said, you can see all these arrows going in the middle of the figure. Obviously there is some data copying and data shuffling. And that is actually one of the shortcomings or constraints of MapReduce. It's a lot of data movement and communication in there.

Old exam question

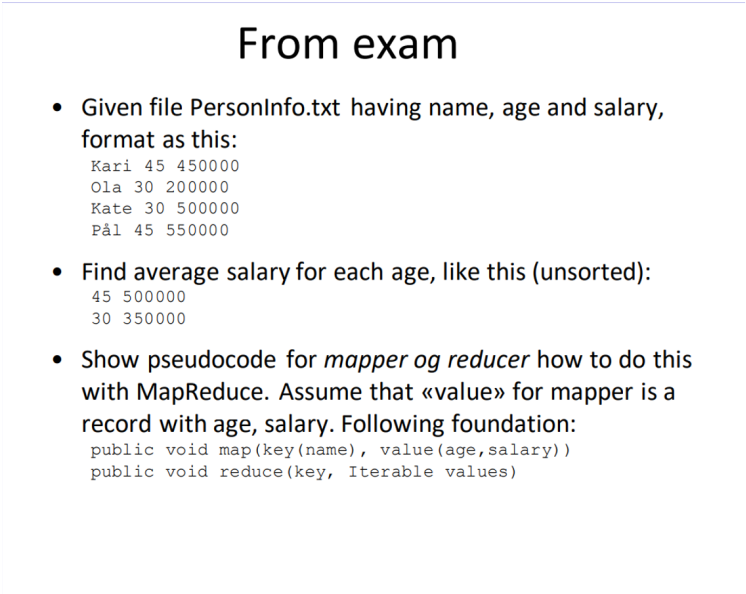


Figure 15: Question

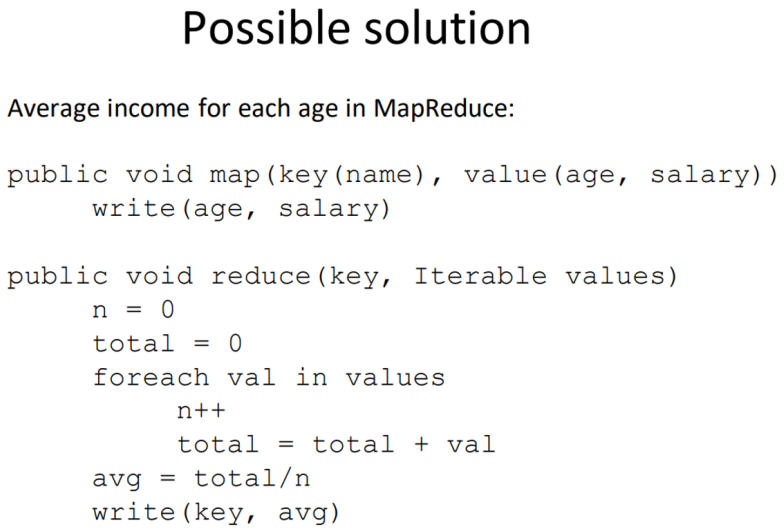


Figure 16: Solution

MapReduce/Hadoop: 6 stages

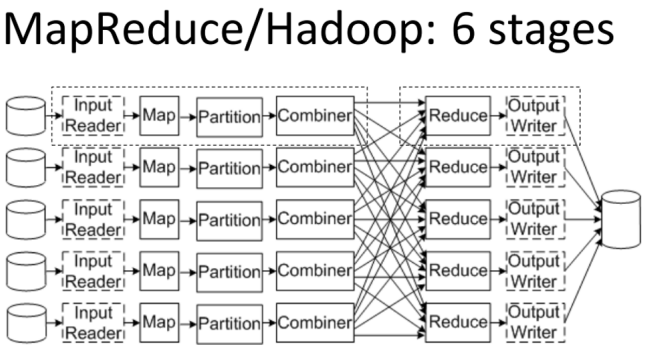


Figure 17: 6 Stages

In Hadoop there is actually six stages of the system. So there is an input reader. There is the mapper. There is some partitioning. And something that's called a combiner. Then there's the reducer. And you see again all these arrows between the systems are obviously because the reducer is assuming data to be in certain grouping. And there is a class of an output writer. This one looks like it's writing to a single drive. But if this is an HDFS drive. This is inherently all sprayed out on multiple machines.

Input Reader

The Input Reader in MapReduce frameworks is a component responsible for loading data and converting it into key-value pairs for further processing. While often used for reading from files, it's adaptable to other data sources, such as databases.

The concept of "splits" is central to this process. A split defines the portion of data that a single map task will process. The size of a split is typically equivalent to a block of data, which is a fundamental unit of storage in a filesystem, and is essential for determining the level of parallelism in the mapping process. For instance, in a file-based data source, a split could correspond to a single file block, whereas in a database, it might relate to a subset of a table.

By configuring splits appropriately, you can control the degree of parallelism: each split can be processed by different map tasks concurrently, thus exploiting the benefits of parallel computation. Having only a single split, however, would mean that only one map task is used, negating the advantage of distributed processing. Hence, it's crucial to ensure that data is appropriately split to maximize efficiency and speed up the map phase of the MapReduce job.

Map-Function

The Map function is a core component of the MapReduce process, receiving key-value pairs from the Input Reader. It processes each pair and can generate multiple new key-value pairs. The outputs are temporarily stored in memory for efficiency and written to disk only when necessary to manage memory usage, a process referred to as "spilling."

If a Map task fails, the system is robust enough to restart that task on the same data segment, ensuring reliability and fault tolerance. This is a key feature of MapReduce, allowing it to handle large data sets and recover from errors without losing progress. However, reprocessing can be time-consuming for large data segments, so there's a trade-off between the granularity of the data processed by each Map task and the system's recovery speed after a failure.

In summary, the Map function:

1. Accepts key-value pairs from the Input Reader.
2. Executes the mapping operation, generating new key-value pairs.
3. Stores the output in memory, spilling to disk when necessary.
4. Supports fault tolerance by restarting tasks upon failure, providing resilience but with potential reprocessing time implications for large data segments.

Partitioning Function

The Partitioning function in MapReduce is responsible for distributing the output from the Map tasks to the Reduce tasks. By default, this function applies a hash function to each key, which determines how the map outputs are assigned to reducers, facilitating load balancing across the nodes.

However, there are scenarios where a custom partitioning strategy is more advantageous. This could be due to the specific characteristics of the data or the desired outcome of the Reduce phase. For instance, certain keys may need to be processed together to optimize the computational efficiency or to comply with the requirements of the task at hand.

The partitioning function is, therefore, a flexible component that can be adjusted or overridden by the user, allowing for tailored distribution of tasks according to the specific demands of the data or the computation logic. This user-defined function can harness insights from the dataset to optimize the processing and achieve more effective results.

Combiner

The Combiner is an optional component in the MapReduce framework that serves as a mini-reducer during the Map phase. It operates on the output of the Map tasks, aggregating intermediate key-value pairs with the same key, which can significantly reduce the amount of data to be transferred across the network to the Reducers.

The Combiner is particularly useful when: - There are many pairs with the same key produced by the Mapper. - The reduction operation is both commutative (the result is the same regardless of the order of the operands, i.e., a+b = b+a) and associative (operations can be regrouped without changing the result, i.e., (a+b)+c = a+(b+c)).

By performing partial combination, the Combiner helps in minimizing the data shuffling phase's communication overhead, leading to more efficient processing in the Reduce phase. Often, the same

function can be employed for both the Combiner and the Reducer, simplifying the development process. However, since it's an optimization step, its implementation depends on the specific use case and the nature of the data and operations involved.

Reduce-function and Shuffle/Sort Details

The Reduce function is a pivotal stage in the MapReduce workflow, invoked once for every unique key generated during the Map phase. It processes all the values associated with a particular key as a single group. By the time data reaches the Reduce phase, it is already sorted by key, which streamlines the reduction process.

If the default key sorting does not fit the requirements—for instance, if the data needs to be sorted by a different criteria or in a non-standard order—a custom comparer can be implemented to define the desired sorting behavior.

Additionally, the shuffle and sort phase plays a critical role in preparing data for the Reduce function. After the map tasks are executed, the outputs are buffered in memory, partitioned, and sorted. This data is then written to disk. Since the data from each map task is already sorted, the subsequent merge step needed before reduction is greatly simplified. The two-stage sorting—first per map task, then merging these pre-sorted chunks during the shuffle—ensures that the reduce tasks receive data that is sorted globally across all map outputs. This efficient preparation is crucial for the effectiveness of the Reduce function.

Output writer

The final component in the MapReduce process is the Output Writer, which handles the storage of computed results. Its primary function is to write the output from the Reduce tasks into durable storage, typically a disk system.

By default, the Output Writer organizes results into a single directory, creating one file for each Reduce task. This default behavior ensures that the output is systematically stored and easily retrievable. However, it is also customizable to cater to different storage needs or integration requirements. Modifications can be made to:

1. Consolidate outputs into a single file, if required.
2. Interface and store the results directly into other systems, such as databases.

This flexibility is crucial for tailoring the output to suit various operational contexts and makes the MapReduce framework adaptable for a wide range of real-world applications, where results may need to be consumed by different downstream systems or stored in a format that is most suitable for the intended use case.

Weaknesses/Limitations of MapReduce

MapReduce, while revolutionary for its time in processing big data, does have notable limitations which have inspired the development of more advanced systems like Apache Spark. Some of the weaknesses of MapReduce include:

1. High Communication Costs: The shuffle and sort phase can be a bottleneck due to the heavy communication load, which can be costly in terms of performance and resources.

2. Lack of Support for Certain Operations:
    - Iterative Computing: Essential for algorithms that require multiple passes over the data, such as many machine learning algorithms.
    - Early Termination: Operations like finding the top K items are not straightforward because MapReduce is designed for complete data processing.
    - Real-time Processing: MapReduce is inherently a batch processing system and doesn't support real-time data processing.
    - Multi-path Operators: Complex operations like joins are not directly supported and require additional programming effort.
3. Designed for Batch Processing: MapReduce is optimized for batch jobs and forward progress through stages, not for interactive or ad-hoc queries.

Despite these limitations, MapReduce has been a foundation for big data processing and has paved the way for the development of more advanced and efficient systems. It has been a catalyst in the big data movement, prompting active research and development to optimize its components, especially the shuffle and sort phases.

In the industry, significant efforts are underway to improve MapReduce and its ecosystem, with initiatives like:

- Enhancing data indexing for faster access.
- Developing methods for early termination to save on computing resources.
- Optimizing resource allocation using technologies like YARN.
- Expanding into stream processing for more real-time data handling.

Such research aims to refine data processing stages and introduce cost savings and performance enhancements. MapReduce's framework continues to evolve, reflecting ongoing innovations and industry needs in big data processing.

YARN

YARN, which stands for "Yet Another Resource Negotiator," is a significant evolution of Hadoop's compute platform, addressing several limitations of the earlier MapReduce system known as MRv1.

In Hadoop 1.x, the JobTracker was the central authority that managed the scheduling and monitoring of tasks across all DataNodes. This framework faced scalability issues as the JobTracker became a choke point in large clusters, limiting the number of concurrent tasks due to its centralized nature.

MRv1 had several specific drawbacks:

- Scalability: The JobTracker had to manage all aspects of job scheduling and task coordination, which became inefficient as the number of tasks and the size of data grew.
- Resource Utilization: Resources were statically allocated, with a fixed number of mappers and reducers per node, leading to underutilization.

- Flexibility: The system was optimized for MapReduce jobs, with limited support for other types of compute tasks.

YARN was introduced to overcome these challenges, decoupling the resource management and job scheduling/monitoring roles that were combined in the JobTracker. In this new architecture:

- The ResourceManager handles the allocation of computing resources across all applications.
- The ApplicationMaster manages the execution of a single application, such as a MapReduce job or a Spark job, allowing for a dynamic and flexible use of cluster resources.
- The NodeManager monitors the resource usage in each node and reports to the ResourceManager.

This redesign allows YARN to:

- Scale more effectively, as the ResourceManager can handle a larger number of applications.
- Utilize cluster resources more efficiently, as it can dynamically allocate resources based on demand rather than being fixed.
- Support a wider variety of computation paradigms beyond MapReduce, including Spark and Tez, essentially acting as an operating system for the Hadoop cluster.

The introduction of YARN has enabled Hadoop to transition from a single-use data processing platform (MapReduce) to a multi-use platform capable of handling a diverse set of data processing and computation models. This makes it possible for different applications to coexist and run simultaneously on a Hadoop cluster, maximizing resource utilization and expanding the cluster's capabilities.

YARN Architecture

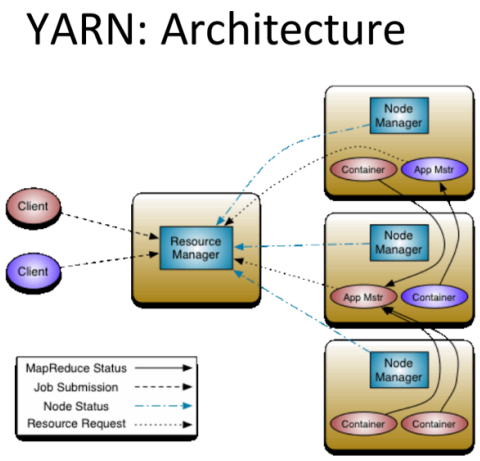


Figure 18: YARN Architecture

1. 1. Resource Manager (RM): The central authority of the system, which handles the allocation of computation resources across the entire cluster. The Resource Manager has two main components:
    - Scheduler: Responsible for allocating resources to various running applications according to policy constraints (like capacity, queues, etc.).
    - Application Manager: Manages the entire lifecycle of applications and maintains a list of submitted jobs.
2. Node Manager (NM): A per-machine framework agent that is responsible for containers, monitoring their resource usage (CPU, memory, disk, network), and reporting the same back to the Resource Manager.
3. Application Master (AM): Each application (like a MapReduce job) has its own Application Master instance responsible for negotiating resources with the Resource Manager and working with the Node Manager to execute and monitor tasks.
4. Containers: These are the basic units of work execution that encapsulate the runtime environment for an application or task. A container may run tasks from MapReduce or any other computing framework compatible with YARN, like Apache Spark.

Summary: YARN vs MapReduceV1

YARN presents a significant leap over MapReduce Version 1 (MRv1), offering numerous advantages that cater to the needs of modern big data processing:

1. Scalability:
    - YARN's architecture, with an individual ApplicationMaster for each job, contrasts with MRv1's single JobTracker. This design alleviates bottlenecks and allows YARN to scale to handle a larger number of concurrent jobs.
2. Increased Availability:
    - By decoupling the ResourceManager and ApplicationMaster, YARN avoids the single point of failure inherent in MRv1's JobTracker. This separation enhances the overall reliability and uptime of the system.
3. Better Resource Utilization:
    - YARN dynamically allocates resources based on demand, leading to more efficient use of the cluster. MRv1 relied on a static allocation model with fixed slots for Map and Reduce tasks defined in the cluster configuration, which could lead to underutilization.
4. Multi-tenancy:
    - YARN supports running various applications and frameworks alongside traditional MapReduce jobs, thus enabling a single cluster to be shared across multiple tenants. This capability is crucial for organizations that run a diverse set of workloads and need to maximize the usage of their hardware investments.

5. Flexible Scheduling:

    - With multiple scheduling options like FIFO, Capacity, and Fair Scheduling, YARN can cater to different organizational policies and workload characteristics. MRv1 did not offer this flexibility, adhering to a more rudimentary job scheduling approach.

In essence, YARN enhances Hadoop's processing abilities by providing a more robust, flexible, and scalable foundation for big data workloads, allowing for improved cluster management, better fault tolerance, and the ability to adapt to varying job demands.

Lecture 4: Spark

Introduction to Spark

Apache Spark emerged as an evolutionary step beyond MapReduce in the domain of Big Data processing. While MapReduce was revolutionary, providing a robust framework for processing vast datasets across distributed systems, it had its limitations. There were several data processing patterns and applications that MapReduce was not equipped to handle efficiently, particularly those that required frequent data reuse.

Spark was conceived out of the need to fill these gaps. It was designed to support a broader spectrum of data processing tasks — from batch processing to stream processing and complex iterative algorithms often used in machine learning and graph processing. The aim was to maintain the core strengths of MapReduce, such as fault tolerance and scalability, while introducing greater flexibility and speed.

One of the critical advancements that Spark brings to the table is its in-memory data processing capability, which allows for faster execution of certain types of computations, particularly those that are iterative or interactive in nature. This feature is especially beneficial in scenarios like interactive data analysis, where a user may need to query and visualize large datasets in real-time, or in iterative algorithms where the same dataset is processed multiple times, such as in machine learning.

Spark also improves upon the developer experience. It provides a rich API that is more expressive and easier to use, which was further enhanced by its integration with the Scala programming language. Scala offers a concise and powerful syntax which is a good fit for Spark's functional programming model. Furthermore, Spark supports multiple other languages like Python, Java, and R, making it accessible to a wider range of developers.

In summary, Apache Spark was developed as a more flexible and efficient successor to MapReduce, able to handle a wider array of data processing tasks with improved speed and developer experience. As Spark continues to evolve, it remains at the forefront of the Big Data processing landscape, providing powerful tools for the next generation of data-driven applications.

Resilient Distributed Datasets

At the heart of Apache Spark's ability to process Big Data at scale is a fundamental data structure known as the Resilient Distributed Dataset (RDD). An RDD is an immutable, partitioned collection of objects, which can be created from a variety of sources including HDFS or parallelized collections. RDDs enable Spark to handle large-scale data processing by enabling transformations through operations like map, filter, and groupBy.

The power of RDDs lies in their resilience; they have the ability to recover from failures swiftly. Each RDD is capable of rebuilding itself using lineage information if a part of it is lost, ensuring that data processing can continue smoothly in the event of a node failure in the cluster.

RDDs are designed to be used in complex multi-stage data processing pipelines, where data is transformed across various stages. They also serve as a buffer for reusing intermediate results across multiple computations, which is particularly beneficial in iterative algorithms that are common in machine learning.

Another crucial aspect of RDDs is their lazy evaluation. Computations on RDDs are only triggered when an action (such as count, reduce, collect, or save) is invoked. This approach allows Spark to optimize the entire sequence of transformations, leading to more efficient execution by computing only what is necessary.

Programming with RDDs can be done using various interfaces provided by Spark, such as the Spark Shell for Scala, PySpark for Python, and Spark SQL for structured data processing. Additionally, Spark provides APIs for R, enabling statistical computing and graphics. These interfaces cater to a wide array of users, from data scientists to application developers, and enhance the accessibility and usability of Spark for Big Data processing tasks.

Comparing MapReduce and Spark Processing

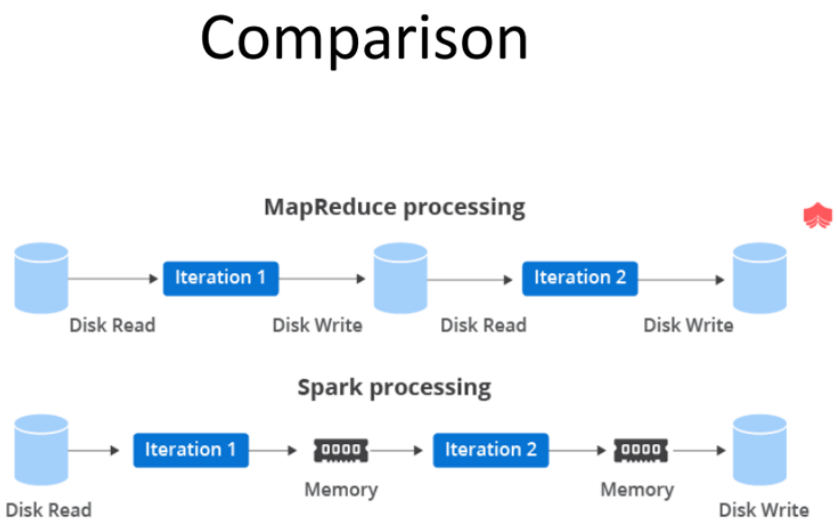


Figure 19: Comparison MapReduce and Spark Processing

In the landscape of Big Data processing, MapReduce and Spark are both pivotal, yet they offer distinctly different approaches. The essence of MapReduce is in its simplicity—data is read from disk, processed through the map and reduce phases, and the results are written back to disk. Each job in MapReduce is discrete and must complete before the next begins, whether it is a subsequent iteration or a different job entirely. This leads to a clear demarcation between each job, with the overhead of reading from and writing to disk for each stage.

Contrastingly, Spark introduces a more dynamic and flexible framework. It enables the chaining of multiple processing stages, significantly enhancing efficiency. With Spark's Resilient Distributed Datasets (RDDs), data can reside in memory across these stages, bypassing the need for disk I/O between intermediate steps. This in-memory processing capability of Spark allows for an iterative and interactive computing paradigm, which is especially beneficial for workloads such as machine learning algorithms that require multiple passes over the same data.

Moreover, Spark's architecture allows for different stages of processing to be interleaved and pipelined, rather than waiting for one job to complete before initiating another. This results in a major leap in efficiency, making Spark more applicable to a new class of data-intensive applications that demand faster processing times and more complex data workflows.

The comparison between MapReduce and Spark highlights Spark's advancements in speed and flexibility, positioning it as an essential tool for handling modern Big Data processing needs that require agility and rapid insights.

Driver vs. Workers

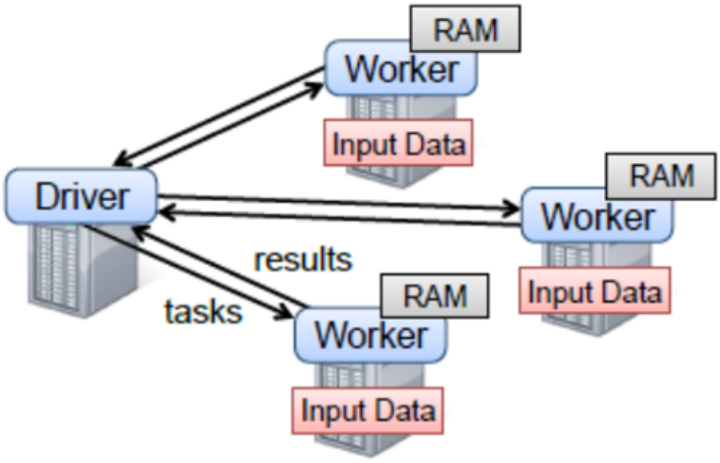


Figure 20: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory

In Spark there are two classes of processing nodes; the main program, called the driver, orchastrating the actions taken, and the workers, which perform all these small actions on the datasets. The framework, thus, knows the topology and how to effectively route the next workers onto actions that are called on the datasets.

Fault Tolerance RDDs

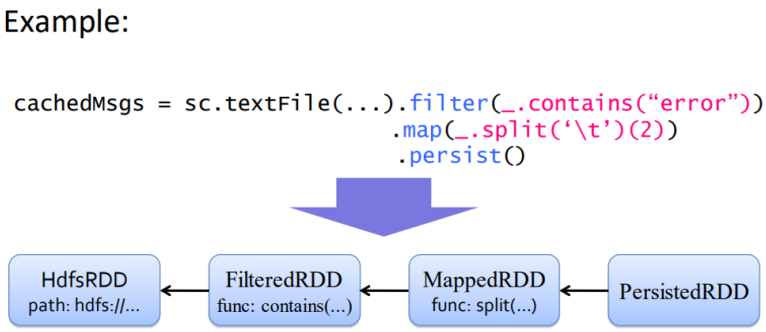


Figure 21: Example

In Apache Spark, fault tolerance is achieved via a concept known as lineage, which is embedded within Resilient Distributed Datasets (RDDs). Lineage is the metadata that Spark maintains for each RDD, detailing the series of transformations that were applied to create it from one or more parent RDDs or from an external data source, such as a file in HDFS.

Lineage is Spark's mechanism for reconstructing an RDD if any part of its data is lost. For example, consider a multi-stage data processing workflow that involves reading a text file from HDFS, applying a filter transformation, then a map transformation to parse tab-separated values, and finally persisting the transformed data. Each stage generates a new RDD, with the map RDD containing a pointer to the filter RDD from which it was derived. In turn, the filter RDD points back to the original text file RDD.

Should an RDD be lost due to a node failure or other issue, Spark uses this lineage information to recompute the RDD from its parent(s). If necessary, Spark can backtrack through the entire lineage chain, right to the original data source, to reconstruct any lost RDD. This recursive reconstruction ability ensures that Spark's data processing pipelines are robust against failures, without the need to checkpoint or replicate the entire dataset at each stage, which would be resource-intensive.

Essential Transformations: Map and Filter

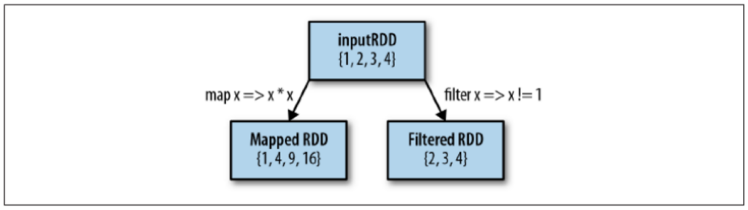


Figure 3-2. Mapped and filtered RDD from an input RDD

Example 3-26. Python squaring the values in an RDD

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print "%i " % (num)
```

Figure 22: Transformations: Map and Filter

In Spark, there's a bunch of transformations, actions you can do. Map and filter are quite essential ones. Map basically takes one input value and convert it to a different one. An example shown here is mapX is the square of X. So for this one, if you have an input RDD, just an array of numbers, the output or the mapped array is just the square of those numbers. A filter is, as you can guess from the verb, removing things that's not fitting a criteria. This one filters, it passes through everything that's not one. So it will remove the first element. And here you can see the Python example for doing this. In Python, we use lambda functions.

Map vs. Flatmap

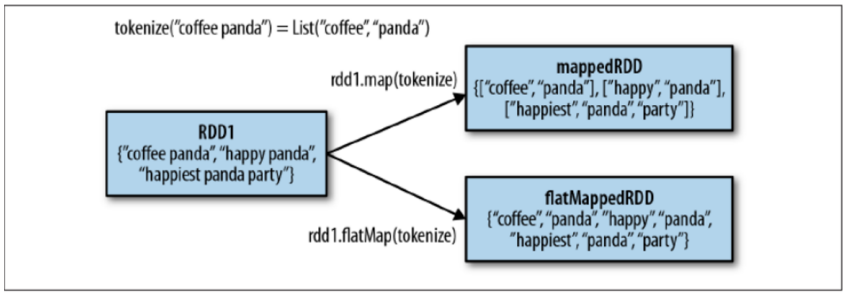


Figure 3-3. Difference between flatMap() and map() on an RDD

Example 3-29. flatMap() in Python, splitting lines into words

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first()  # returns "hello"
```

Figure 23: Map vs Flatmap

Another concept is mapping and flatmapping. There is the standard map that takes one input item and generates one output item. So this one is tokenizing, basically splitting strings up by white spaces or tabs or whatever. So coffee panda is converted into coffee, the word, and panda, the word. So it's generating the words. But it generates one array per input item. The flat map completely, from the name, flattens the structure. From one input, it can generate arbitrary number of outputs. So this can be useful if you're reading hierarchical structure data and you want to flatten everything out to count words or whatever. This one completely removes the structure.

Reduce

Example 3-32. reduce() in Python

```
sum = rdd.reduce(lambda x, y: x + y)
```

Figure 24: Reduce in Python

Aggregates elements in dataset by using a function f taking in an arbitrary amount of arguments and returning one.

RDD Transformations and Actions

Expected that we know these for the exam:

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	`9`

In general, all the actions in Spark keep the order except the ones that doesn't and those are obvious

Key/Value-pairs

Spark also has special love in its system for key value pairs. So, that's a special basic data type. Key value pairs you can make from just retrieving from a file that has special formatting like CSV, TSV, comma separated and tab separated files. You can also create pairs from a map function. Pairs are just described as an array with a comma in it.

For this basic datatype we have more functions that we need to know:

Table 1: Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey((x, y)=> x + y)`	{{(1, 2), (3, 10)}}
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	{{(1, [2]), (3, [4, 6])}}
`values()`	Return an RDD of just the values.	`rdd.values()`	{2, 4, 6}
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`	{{(1, 2), (3, 4), (3, 6)}}
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	{{(1, 3), (3, 5), (3, 7)}}
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5))`	{{(1, 2), (1, 1), (3, 4), (1, 5), (3, 5)}}
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	{1, 3, 3}

Table 2: Transformations on two pair RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	{{(1, 2)}}
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	{{(3, (4, 9)), (3, (6, 9))}}
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.rightOuterJoin(other)`	{{(3, (Some(4), 9)), (3, (Some(6), 9))}}
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.leftOuterJoin(other)`	{{(1, (2, None)), (3, (4, Some(9))), (3, (6, Some(9)))}}

Table 3: Actions on pair RDDs (example {(1, 2), (3, 4), (3, 6)})

Function	Description	Example	Result
`countByKey()`	Count the number of elements for each key.	`rdd.countByKey()`	{{(1, 1), (3, 2)}}
`collectAsMap()`	Collect the result as a map to provide easy lookup.	`rdd.collectAsMap()`	Map{(1, 2), (3, 4), (3, 6)}
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	[4, 6]

RDD Persistency Levels

In Apache Spark, managing the persistence of RDDs is critical for optimizing performance, especially when dealing with iterative algorithms or workflows where the same data is accessed multiple times. Spark provides several persistence levels to control how and where RDDs are stored, allowing for trade-offs between memory usage, CPU time, and fault tolerance. The simplest form of persistence is MEMORY ONLY, where RDDs are stored in the memory of worker nodes. This approach offers the fastest access times but at the cost of higher memory usage and the risk of losing the RDD if a node fails and there's not enough memory to hold the data. Conversely, DISK ONLY persistence ensures that the RDDs are stored only on disk, which means slower access times but lower memory consumption and no risk of data loss if memory runs out. The MEMORY AND DISK level attempts to balance the two by storing the RDDs in memory first, but if the memory is insufficient, it spills the excess data to disk. This provides a good balance between speed and data safety, ensuring that data is not recomputed even if it doesn't fit entirely in memory. Serialization can be applied to the objects in the RDD to save space (MEMORY ONLY SER, MEMORY AND DISK SER), converting them into byte arrays. While serialization reduces memory usage, it does add computational overhead, as objects need to be deserialized before they can be processed. Therefore, serialized storage levels are a trade-off between space and computation time. For ease of use, Spark also provides the RDD.cache() method, which is a shorthand for RDD.persist(MEMORY ONLY). This method is syntactic sugar for the common pattern of keeping an RDD in memory for faster access. Choosing the right persistence level is an essential decision in Spark programming and is typically guided by the specific requirements of the workflow. For RDDs that will be reused promptly and fit comfortably in memory, MEMORY ONLY is ideal. For RDDs that are expensive to compute and will be reused, but may not fit entirely in memory, MEMORY AND DISK is often the best choice. If memory is at a premium and computation is less expensive, then DISK ONLY or a serialized persistence level may be more appropriate.

Representing RDDs

RDDs carry essential metadata as part of their lineage. This metadata maintains information about:

- Partitioning: Defines how the data is divided into chunks or partitions that can be processed in parallel.
- Locations: Specifies the physical location of data partitions, which is particularly relevant when RDDs represent data stored in Hadoop Distributed File System (HDFS).

Efficient Data Retrieval
By knowing the exact locations of partitions, especially when they correspond to blocks in HDFS, Spark can retrieve data efficiently. Each partition typically corresponds to a block in HDFS, enabling Spark to locate and process data with high efficiency.

Partitioning Schemes
Partitioning in Spark can be performed using two primary methods:

- Range Partitioning: Data is divided based on a range of key values, allowing for ordered processing.
- Hash Partitioning: Keys are hashed, and partitions are created based on the hash value, ideal for distributing data evenly across nodes.

Dependency Tracking
RDDs also keep track of their dependencies, that is, other RDDs they are derived from. There are two types of dependencies:

- Narrow Dependency: Indicates a one-to-one relationship between partitions of the parent and the child RDD. Each partition of the parent RDD is used by at most one partition of the child RDD.
- Wide Dependency (Shuffle): A more complex relationship where multiple child partitions may depend on a single parent RDD partition, often requiring a shuffle of data across partitions and nodes.

RDD dependencies are pivotal in understanding the lineage and computational strategies within Spark's execution model. Some transformations and actions, like map operations or union transformations, exhibit narrow dependencies where there is a direct, one-to-one mapping between the input and the output. This streamlined relationship simplifies execution, as computations can be pipelined within a single node, resulting in efficient recovery mechanisms and lower computational costs.

On the other hand, operations like groupBy and join, particularly without co-partitioning, establish wide dependencies. Such dependencies introduce a different cardinality, necessitating a shuffle of data across different nodes. While this adds complexity to the task's execution, it significantly enhances the computational capabilities, allowing for more intricate and powerful data processing operations.

Spark Execution

Laziness in Execution

Apache Spark operates on a *lazy execution* model, constructing a Directed Acyclic Graph (DAG) that outlines all the operations to be performed but defers their execution. The actual processing is only triggered by actions that return results to the driver program or write data to external storage systems.

Stages and Dependencies

The computation in Spark is broken down into stages, which are formed based on the presence of **wide dependencies**. These dependencies, which often necessitate shuffling data across the cluster, define the boundaries of each stage. Within a stage, tasks operate on narrow dependencies that can be processed in parallel on a single machine.

Stages: A stage is a collection of tasks that can execute in parallel, determined by the shuffle boundaries required by wide dependencies.

RDDs and Tasks: Resilient Distributed Datasets (RDDs) are represented as rectangular boxes in the execution graph, with smaller rectangles indicating partitions. Shaded partitions represent those already in memory. Tasks are the smallest units of work in Spark and are associated with processing a single partition of an RDD.

Execution Hierarchy

Application The top-level unit of execution, representing a user program that coordinates the execution of jobs.

Jobs Sequences of stages triggered by actions, each representing a computation resulting from an action.

Stages Divisions within a job, corresponding to the computational boundaries where data is shuffled.

Tasks Individual units of work that execute code on different partitions of data across the cluster.

Cross-Machine Execution

Data shuffling between stages can occur within a single machine or across the cluster. While in-memory shuffling is possible, Spark typically relies on HDFS or other distributed storage systems for data shuffling, ensuring reliability and scalability of the distributed processing.

Sprak SQL: Dataframes

Spark SQL is a module in Apache Spark for structured data processing. It provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine.

Introduction to DataFrames

A DataFrame is a distributed collection of data organized into named columns, conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

Features of DataFrames

- **Immutability and Lazy Evaluation:** DataFrames are immutable and lazily evaluated, allowing Spark to optimize the entire query plan from data read to data write.
- **Interoperability:** They provide a domain-specific language for structured data manipulation in Scala, Java, Python, and R.
- **Integration:** DataFrames are deeply integrated with Spark SQL's built-in functions and can be directly used to run SQL queries.
- **Performance:** They offer significant performance improvements over RDDs for structured data processing due to Spark's Catalyst optimizer and Tungsten execution engine.

DataFrame Operations

DataFrame operations are divided into transformations and actions. Transformations, such as selecting and filtering, create a new DataFrame from an existing one. Actions, such as collecting the data to the driver, trigger the execution of the computation.

DataFrame API

The DataFrame API in Spark SQL provides a variety of functions to perform complex operations such as aggregation, joining, and window functions in a concise manner. It's designed to simplify the development process by providing a higher level of abstraction over RDDs.

Execution Plan Optimization

When an action is called on a DataFrame, Spark SQL uses the Catalyst optimizer to generate an optimized logical and physical execution plan. This optimization includes predicate pushdown, column pruning, and other rule-based optimizations.

Usage Example

Here's an example of creating a DataFrame from a JSON file and performing a transformation and an action:

```
val df = spark.read.json("path/to/jsonfile")
df.filter($"age" > 21).select("name", "age").show()
```

This snippet reads a JSON file into a DataFrame, filters the records for age greater than 21, selects the name and age columns, and finally displays the result.

Spark on YARN

Apache Spark is designed to run on various cluster managers, and one of the most common integrations is with YARN (Yet Another Resource Negotiator). YARN is Hadoop's cluster management system, and Spark can run on YARN to utilize its resource management capabilities.

YARN Integration Benefits

The integration of Spark with YARN allows Spark to benefit from YARN's features such as:

- The **NodeManager**, which manages the resource usage in each node.
- The **ResourceManager**, which coordinates the allocation of compute resources.

Modes of Operation

There are two primary modes for running Spark on YARN:

Interactive Mode: In interactive mode, often used for development and debugging, the Spark driver runs on the client's machine. The cluster, which could range from a single machine to a large number of nodes, acts as the worker nodes or executors.

Cluster Mode: In production, a minimal client stub is used to submit the job to the cluster. The Spark driver then runs inside the cluster on a YARN NodeManager node. This setup includes:

- A **Spark Application Manager** that resides within the cluster.
- A set of **executor backends** that manage the worker nodes and handle task execution.

Finding Similar Items

Finding similar items in big data is a crucial task in data analysis and machine learning. It involves identifying items that share common characteristics or patterns within massive datasets, enabling tasks such as recommendation systems, clustering, and anomaly detection. Techniques like locality-sensitive hashing, minhashing, and clustering algorithms are commonly employed to efficiently uncover similarities amidst the vast amount of data, facilitating tasks ranging from e-commerce product recommendations to fraud detection in financial transactions. We will focus on locality-sensetive hashing in this course.

Introduction to High-Dimensional Data Points

In many applications, data can be represented as high-dimensional vectors. These vectors could encapsulate a variety of data types, including strings, pixels, or textual information. Crucially, a distance function $d(\vec{x}_1, \vec{x}_2)$ is needed to quantify how different or similar two data vectors, $\vec{x}_1$ and $\vec{x}_2$, are. This function is fundamental in determining the usefulness of the data representation.

Problem Statement: Given a set of high-dimensional data points $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N$, the goal is to find all pairs of points $(\vec{x}_i, \vec{x}_j)$ that are within a specified distance threshold $d(\vec{x}_i, \vec{x}_j) \leq s$, thereby identifying similar items.

Naïve Approach: A straightforward approach would be to compute the distance between every possible pair, which would be on the order of $O(N^2)$. This method, albeit simple, is computationally expensive and inefficient. It is also worth noting that since distance measurements are typically symmetric, it is sufficient to compute only half of the possible pairs.

Efficient Solution: The lecture will explore advanced methods that significantly reduce the computational complexity. Instead of the quadratic time complexity of the naïve approach, we will

demonstrate a strategy that can identify similar items in linear time, $O(N)$, which is a substantial improvement for large datasets typical in the realm of Big Data.

Distance Measures

An essential aspect of finding similar items is defining the concept of distance within the high-dimensional space. The term "neighbors" is formalized to mean points that are within a "small distance" from each other. However, the definition of "distance" can vary depending on the specific application and the nature of the data.

Jaccard Distance and Similarity

One intuitive measure of distance is the Jaccard distance, which derives from the concept of Jaccard similarity. The Jaccard similarity between two sets is the ratio of the magnitude of their intersection to the magnitude of their union, mathematically defined as:

$$\text{sim}(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

Accordingly, the Jaccard distance is defined as one minus the Jaccard similarity:

$$d(C_1, C_2) = 1 - \text{sim}(C_1, C_2)$$

This normalized metric, which ranges between 0 and 1, intuitively represents how dissimilar two sets are, with 0 indicating identical sets and 1 indicating no common elements.

Example: Considering two sets represented by circles in a Venn diagram, the intersection contains the elements present in both sets, while the union encompasses all elements from both sets. The Jaccard similarity is then the proportion of the intersection relative to the union. It is important to note that sets without common elements do not contribute to the intersection and thus do not increase the denominator in the Jaccard similarity calculation.

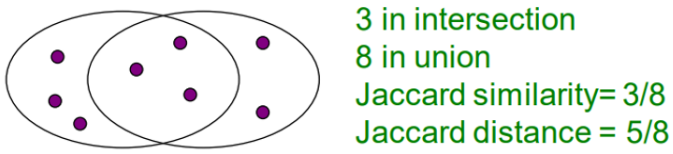


Figure 25: Illustration of Jaccard similarity and distance with two overlapping sets.

Task: Finding Similar Documents

Objective: In the realm of Big Data, a pertinent task is to manage and analyze a vast quantity (N) of documents that may range into the millions or billions. The primary goal is to identify pairs of documents that are "near duplicates" of each other.

Applications

The techniques discussed in this lecture have practical applications in various domains, including:

- Identifying mirror or approximate mirror websites, which are undesirable to display concurrently in search engine results.
- Clustering similar news articles from multiple sources that report the same story, thereby enhancing the organization and retrieval of information.

Challenges

However, several challenges arise in this task:

- Documents may contain many small fragments of content that appear in a different order in other documents, complicating the detection of similarities.
- The sheer number of documents makes it impractical to compare all possible pairs due to computational and memory constraints.
- The size of individual documents or the overall quantity may be so extensive that they cannot be processed in the main memory of standard computing systems.

The subsequent sections will address how we can tackle these challenges by leveraging specific techniques to efficiently find similar documents without the need for $O(N^2)$ computations, which is infeasible for databases containing billions or trillions of documents.

The Three-Step Process for Similarity Detection

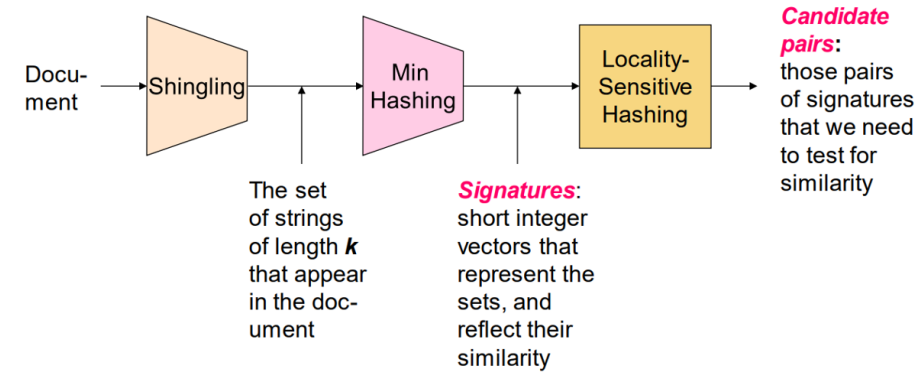


Figure 26: The three-step process of finding similar documents: Shingling, Min-Hashing, and Locality-Sensitive Hashing.

The procedure for detecting similar documents can be divided into a three-step process, each serving a unique purpose in the overall task.

Shingling

Shingling is the initial step where we transform documents into a representative set of identifiers. These identifiers, or shingles, consist of contiguous sequences of k items (such as words or characters) from the document. The choice of k and the items used can significantly affect the subsequent steps in similarity detection.

Min-Hashing

After shingling, we apply a technique called min-hashing. Min-hashing condenses the large set of shingles into a more manageable representation known as signatures. These signatures are short integer vectors that encapsulate the essence of the document's shingles and preserve their similarity. The min-hash function is carefully chosen to ensure that the probability of a match reflects the similarity between the sets of shingles.

Locality-Sensitive Hashing

The final step is locality-sensitive hashing (LSH), which focuses on efficiently finding candidate pairs of signatures to test for similarity. LSH aims to increase the probability that similar items are hashed to the same buckets, thus reducing the number of comparisons needed to find similar pairs.

Candidate Generation: Through LSH, we generate a list of candidate pairs that are likely to be similar based on their hash signatures. These candidates can then be examined further using a more precise algorithm or inspected by a domain expert, such as a journalist, to confirm their similarity.

Shingling as a Basis for Document Comparison

Converting Documents into Sets: Shingling is the process of transforming documents into sets of tokens, which serve as unique identifiers for the documents' content. Unlike methods that identify documents with similar topics, shingling aims to find documents with very close textual compositions, despite minor variations in structure.

Shingle: A k-shingle, or k-gram, is a sequence of k tokens from the document that may be characters, words, or other units depending on the specific application. For example, given a document D with the text 'abcab', the set of 2-shingles would be $S(D) = \{ab, bc, ca\}$.

Simple Approaches and Their Limitations: One could conceive simple methods of shingling, such as using the set of all words or a set of "important" words in a document. However, these approaches fail to capture the nuances necessary for the application at hand, mainly because they do not account for the order of words which can be crucial for similarity detection.

The Importance of Token Order: Shingling differs from these simplistic approaches by considering the order of tokens, creating shingles that reflect the sequence of content within a document.

Measuring similarity To measure the similarity between documents, we convert the shingle sets into sparse 0/1 vectors in the space of k-shingles, where each unique shingle corresponds to a dimension in this space(i.e. all shingles in our shingel set for a document are set to 1 for a vector

that is the size of the k-shingle set (representing all shingles)). The Jaccard similarity can then be naturally applied to these vectors to calculate the similarity measure.

Working Assumption and Parameters: It is critical to select an appropriate size for k. A small k might make most documents appear similar, especially if they share many common tokens. A general heuristic is to choose $k = 5$ for short documents and $k = 10$ for longer ones, though this parameter should be fine-tuned for each application. The choice of k significantly influences the effectiveness of shingling. It should be large enough to capture the uniqueness of the documents while keeping computational requirements in check.

Practical Considerations for Shingling

In practice, shingling is a balance between computational efficiency and the granularity of textual representation. The selection of k plays a pivotal role in the utility of shingling, with larger k values capturing longer sub-sequences of tokens that might contribute to a more nuanced understanding of text similarity.

MinHashing

Short Explanation and Motivation

MinHashing represents a pivotal step in efficiently estimating the similarity between documents in large datasets. The primary goal of this technique is to reduce the high-dimensional data obtained from shingling into a more compact form without significantly sacrificing the accuracy of similarity measures. Considering the computational challenge posed by near-duplicate detection among N documents, the naive approach to computing Jaccard similarities for every document pair is not feasible. For instance, with $N = 1$ million documents, this method would require approximately 5×10^{11} comparisons. Even at a rate of one million comparisons per second, this task would consume an inordinate amount of time, making it impractical for real-world applications.

Encoding Sets as Bit Vectors

A crucial step in efficient similarity computation is the encoding of sets as bit vectors, which allows us to use bitwise operations for quick computations. In this representation, each set is transformed into a 0/1 vector, with each dimension representing an element in the universal set. This encoding facilitates the interpretation of set intersections and unions as bitwise AND and OR operations, respectively.

Example: For instance, given two sets represented as bit vectors $C_1 = 10111$ and $C_2 = 10011$, the intersection (bitwise AND) yields three common elements 10011, and the union (bitwise OR) contains four elements 10111. Thus, the Jaccard similarity is the ratio of the intersection to the union, which is 3/4, and the Jaccard distance is $1 - \text{similarity} = 1/4$.

This bit vector encoding is highly space-efficient for sparse data and is fundamental for the subsequent MinHashing process.

From Sets to Boolean Matrices

When dealing with multiple sets, such as documents represented by shingles, a Boolean matrix becomes an effective data structure. In such a matrix, rows correspond to elements (shingles) (all

the shingles in the universal set), and columns represent sets (documents). A value of 1 in row e and column s indicates that element e is a member of set s.

Column Similarity: The similarity between columns is equivalent to the Jaccard similarity of the corresponding sets. For example, given two columns representing documents, the intersection is the count of rows with value 1 in both columns, and the union is the count of rows with value 1 in either column.

Sparse Matrices: These matrices are typically sparse, with many zeros and relatively few ones. The sparseness is beneficial for computational efficiency, as operations can often be optimized to only consider the non-zero elements.

Document as a Column: Each document is represented as a column in this matrix, with the Jaccard similarity between any two documents quickly computable using bitwise operations on their corresponding columns.

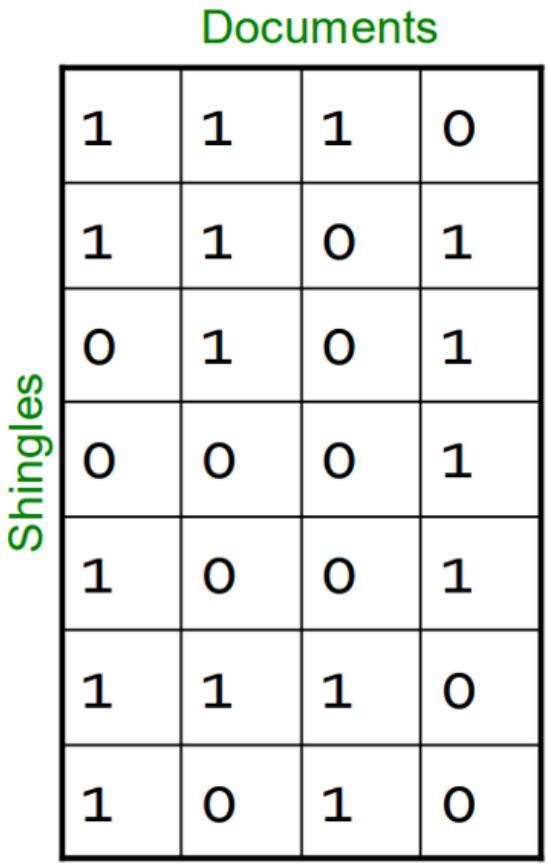


Figure 27: A Boolean matrix representation of documents and their shingles, highlighting the sparse nature of such matrices and the column-wise computation of Jaccard similarity.

Finding Similar Columns via Signatures

Having encoded documents as sets of shingles and represented these sets as boolean vectors within a matrix, the next objective is to identify similar columns—essentially, finding documents with substantial overlap in their shingle sets.

Hashing Columns to Create Signatures

A key aspect of efficiently finding similar documents is to represent each document, not by the full set of its characteristics, but by a concise summary: a signature. The process of hashing columns into signatures involves mapping each document's shingle set to a smaller, fixed-size representation that fits in RAM and reflects the similarity of the original sets.

Designing the Hash Function: The goal is to design a hash function $h(\cdot)$ such that if the Jaccard similarity $\text{sim}(C_1, C_2)$ is high, the probability of $h(C_1) = h(C_2)$ is also high. Conversely, if $\text{sim}(C_1, C_2)$ is low, $h(C_1)$ and $h(C_2)$ should be different with high probability.

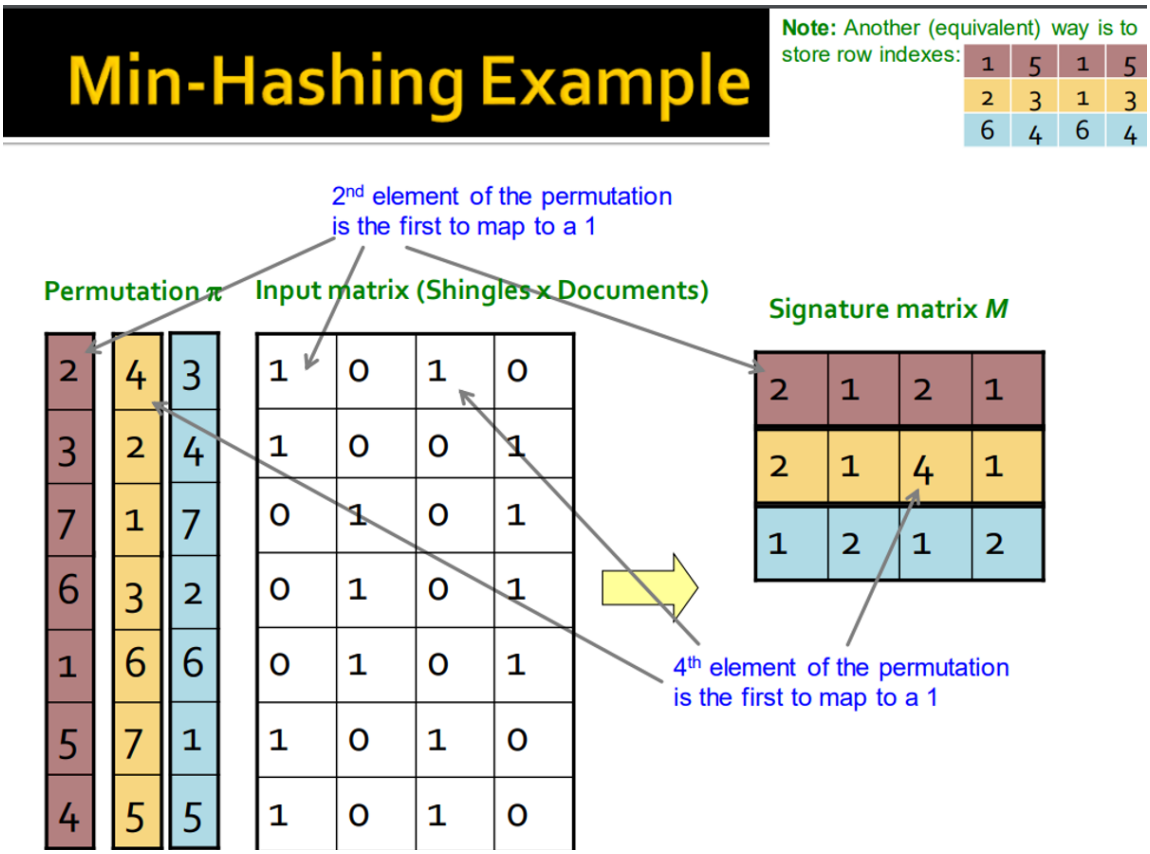
Min-Hashing Technique

The method of Min-Hashing serves as a suitable hash function when working with the Jaccard similarity metric. This technique efficiently estimates the similarity between large sets by interpreting each set as a column of a Boolean matrix.

The Min-Hash Function: Imagine permuting the rows of the Boolean matrix under a random permutation π. The Min-Hash function $h_\pi(C)$ for a column C is defined as the index of the first (in the permuted order π) row in which column C has a value of 1: $h_\pi(C) = min_\pi \pi(C)$.

Creating Signatures: To construct a signature for a column i.e. the whole matrix, we use multiple independent hash functions, corresponding to different permutations, to capture various aspects of the original set. For each permutation we initialize a row with length corresponding to the amount of document sets, with the values "null" not yet determined. Then, for each permuted row, if the column value in that row is 1, we update our initialized row for the signature matrix to have the value of the order we are at in the permutation. When all columns have gotten their smallest value for the permutation we have completed the row for that hash function. We do this procedure for all hash functions (i.e. permutations of the boolean matrix) that we apply. The resulting matrix is the signature matrix corresponding to the minHash values.

MinHash Example:



The example shows our method for constructing the signature matrix based on the hash functions / permutations. We see that it corresponds to our explanation as the first row in the signature matrix is 2121, and for the first row in the permuted matrix for the first permutation we have the value 1 in column 1 and 3, and the second permuted row in 0 and 2. Thus, the first row in the signature matrix should be 2121, which it is.

The Min-Hash Property

- **Choose a random permutation π**
- **Claim:** $\Pr[h_\pi(C_1) = h_\pi(C_2)] = \text{sim}(C_1, C_2)$
- **Why?**
  - Let X be a document (set of shingles), $y \in X$ is a shingle.
  - Then: $\Pr[\pi(y) = \min(\pi(X))] = \frac{1}{|X|}$
    - It is equally likely that any $y \in X$ is mapped to the minimum element.

- Let y be such that $\pi(y) = \min(\pi(C_1 \cup C_2))$.
- Then either:
  * $\pi(y) = \min(\pi(C_1))$ if $y \in C_1$, or
  * $\pi(y) = \min(\pi(C_2))$ if $y \in C_2$
- So the probability that both are true is the probability $y \in C_1 \cap C_2$.
- $\Pr[\min(\pi(C_1)) = \min(\pi(C_2))] = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} = \text{sim}(C_1, C_2)$

Four types of rows

Given two columns C_1 and C_2 for b boolean matrix we can have the rows: 11, 10, 01 and 00, where we call the types a, b, c and d. Note that we do not calculate rows 00 in Jaccard similarity as we focus on comparing the presence of elements (or features) in two sets, rather than their absence. Hence, we have $sim(C_1, C_2) = \frac{a}{a+b+c} = P(h(C_1), h(C_2))$. Note that this is how we find simialrity for columns, by calculating the Jaccard similarity for them.

Similarity for signatures

The similarity of signatures is defined as the fraction of the minhash functions in which they agree. When considering signatures as columns of integers, the similarity is the fraction of rows where the corresponding elements match. Therefore, the expected similarity of two signatures corresponds to the Jaccard similarity of the columns or sets they represent. Moreover, longer signatures lead to a smaller expected error in the similarity estimation.

Implementation Trick for Computing Min-Hash Signatures

Computing Min-Hash signatures requires a clever implementation trick to overcome the computational costs. Directly permuting rows for each hash function is infeasible for large matrices. Instead, we use an additional hash function to simulate the effect of permuting rows without the actual permutation.

One-Pass Implementation: The one-pass implementation proceeds as follows: for each column C in the matrix, and for each hash function k_i, we initialize an array sig(C) with a default value of infinity. We then iterate over the rows of the matrix, updating the signature array with the minimum hash value encountered:

```
for each column C do
  for each hash function k_i do
    sig(C)[i] ← ∞
  end for
  for each row j where j has a 1 in column C do
    for each hash function k_i do
      if k_i(j) < sig(C)[i] then
        sig(C)[i] ← k_i(j)
      end if
    end for
  end for
end for
```

Universal Hashing: To define the hash functions k_i, we use universal hashing, where each hash function $h_{a,b}(x)$ is defined as $((a \cdot x + b) \mod p) \mod N$, with a and b as random integers, and p a prime number larger than N.

Locality-Sensitive Hashing: Finding Similar Signatures

Introduction:

After efficiently generating compact signatures for our documents using Min-Hashing, we now confront the challenge of identifying similar documents without falling back on the computationally prohibitive pairwise comparison. The solution? Enter the third level of hashing: Locality-Sensitive Hashing (LSH).

Beyond Signatures: With signatures in hand, we seek a method to compare them and efficiently identify the candidate pairs—those pairs of signatures that are likely to represent similar documents. A direct comparison of signatures for all possible pairs would still result in an $O(N^2)$ problem, which is not scalable.

Grouping with LSH: Locality-Sensitive Hashing provides an elegant solution to this problem. LSH is a hashing technique designed to hash similar items into the same "buckets" with high probability. The intuition is that if two signatures are similar, they are likely to remain similar under the hash functions used by LSH, thus being bucketed together.

Candidate Generation: By hashing signatures using LSH, we form groups or buckets of signatures that are likely similar, significantly reducing the number of comparisons needed. Instead of comparing every signature with every other signature, we only compare signatures within the same bucket. This process effectively narrows down the candidate pairs to a manageable set that can be examined in detail for actual similarity.

Achieving Linearity: LSH steers us away from the quadratic complexity of the pairwise comparison approach. By cleverly grouping likely similar items, LSH allows us to focus on a much smaller subset of signature pairs, bringing us closer to a linear-time algorithm for identifying similar documents.

Implementation Considerations: The practicality of LSH lies in its ability to create buckets that are good candidates for similar items, facilitating either a manual check or a further automated detailed comparison. This step is crucial for dealing with massive datasets where classical comparison methods would falter under the sheer volume of computations required.

Locality-Sensitive Hashing

The signature matrix M obtained from the Min-Hashing step is substantial in size. To facilitate the Locality-Sensitive Hashing process, we partition M into b bands, each containing r rows. This technique enhances the hashing precision and effectively narrows down the candidate pairs for similarity checks.

Hashing Bands to Buckets: For each band, we hash the corresponding segment of each column into a hash table with k buckets, where k is a large number. By doing so, we seek to ensure that candidate pairs, or column pairs, hash to the same bucket if they are similar according to the band's segment.

Identifying Candidate Pairs: A pair of columns x and y from M is deemed a candidate pair if their signatures agree on at least a fraction s of their rows. Specifically, if $M(i, x) = M(i, y)$ for at least $\frac{s}{r}$ rows within a band, they are considered candidates for having high Jaccard similarity.

Selecting b and r: The selection of b and r is pivotal. The aim is to tune these parameters to catch most similar pairs while avoiding an overwhelming number of non-similar pairs. A larger b implies more bands and, therefore, more detailed hashing, increasing the chances of identifying truly similar pairs.

The Hashing Scheme: As depicted in the hashing schematic, each band's hash function maps columns of M into buckets. Columns that consistently hash into the same bucket across one or more bands are potential matches. For example, if columns 2 and 6 consistently hash to the same bucket across several bands, they are likely to be very similar, making them candidate pairs for further evaluation.

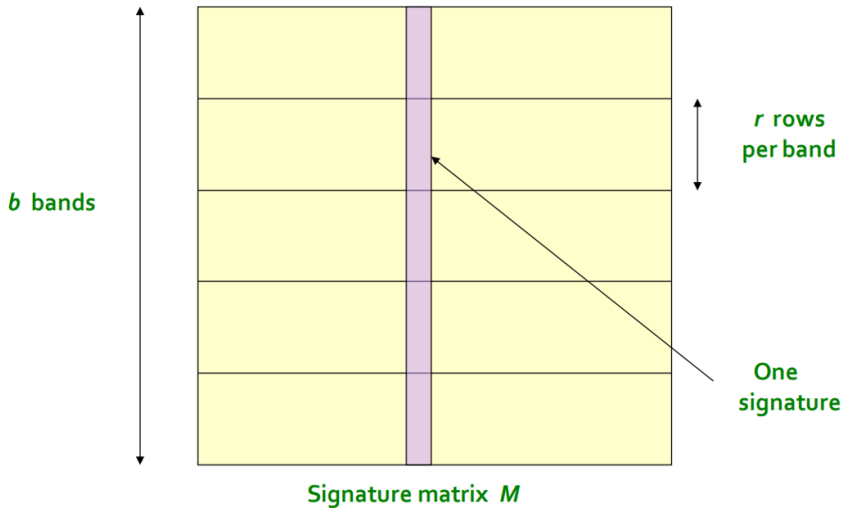
Impact on Similarity Search: This banding strategy of LSH serves to refine the similarity search. It allows us to apply hashing within manageable segments of the signature, making the similarity detection process both granular and efficient. It ensures that we do not miss out on truly similar documents due to overly broad hashing, nor do we get bogged down in evaluating every possible pair.

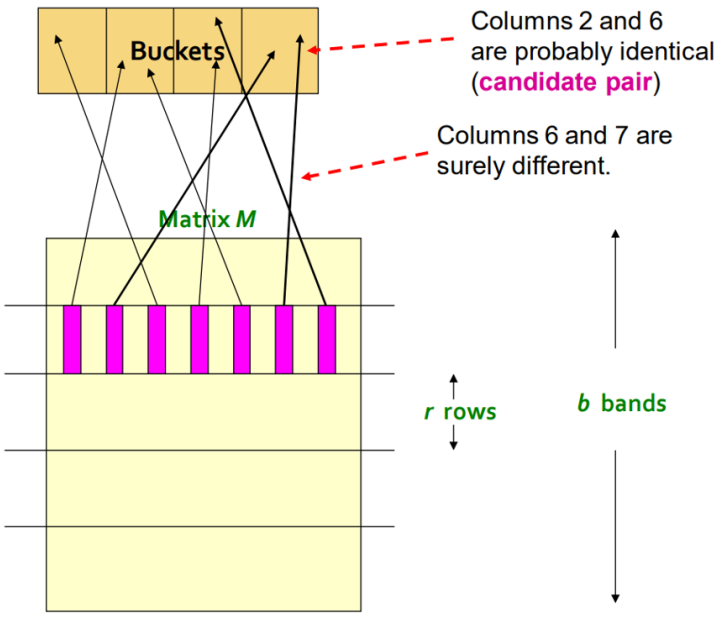
Algorithmic Implementation: Algorithmically, this approach translates into:

- Dividing the signature into bands and initializing hash tables for each band.
- Hashing each segment of the signature corresponding to a band and recording the bucket.
- Declaring columns that hash to the same bucket as candidate pairs for the respective band.

By hashing segments of the signature matrix, we preserve the probability that similar documents remain detectable through hashing, while significantly reducing the number of comparisons needed to identify similar documents in a large dataset.

Illustration:





The Simplifying Assumption in Locality-Sensitive Hashing

Locality-Sensitive Hashing relies on a fundamental simplifying assumption to function effectively. We presuppose the presence of a sufficiently large number of buckets so that the only columns that hash to the same bucket are those that are highly probable to be identical within that specific band. This assumption is not about the correctness of the algorithm but is in place to facilitate the analysis.

Assumption Justification: Given a hashing process over b bands, the likelihood of two non-similar columns colliding in the same bucket should be minimized. Therefore, we aim for a bucket count that makes such collisions highly improbable. While this does not guarantee the correctness of the algorithm, it greatly simplifies the process by which we analyze and understand the behavior of our hashing functions.

Operational Implication: In practice, this assumption leads us to consider that when two columns land in the same bucket, they are 'identical in that band' for the purposes of candidate pair selection. This is a deliberate oversimplification to streamline the analysis, but it does not impact the validity of the overall algorithm, which is robust to the nuances of hash collisions.

This simplification underpins the practical implementation of LSH, allowing us to handle vast datasets with a manageable number of buckets while still maintaining high confidence in the algorithm's ability to identify similar items.

Example

In the given scenario, we have a large collection of 100,000 documents. The task is to identify pairs of documents that are at least 0.8 similar to each other. The process uses a technique known as "Locality-Sensitive Hashing" (LSH) to efficiently find these pairs.

Each document is represented by a signature of 100 integers. These signatures collectively take up 40 megabytes of memory, assuming that each integer requires four bytes of storage. The LSH algorithm segments these signatures into 20 bands, with each band containing 5 integers.

The similarity threshold is set to 0.8, meaning that two documents must share at least 0.8 of their signature in order to be considered similar. The probability that two 0.8 similar documents will hash to the same bucket in any given band is 0.8 raised to the power of 5 (the number of integers in a band), which is approximately 0.328. However, we want to avoid false negatives (similar documents not being identified as such) at all costs, as they represent a failure of the algorithm to accurately identify similar documents. Therefore, the probability that two 0.8 similar documents do not hash to the same bucket in any of the 20 bands is very low $(1-0.328)^{20}$, which is approximately 0.00035. This means there's only a 0.035-percent chance that an 80-percent similar pair of documents will be missed by the algorithm, which corresponds to roughly 1 in 3,000 documents being a false negative.

Conversely, if two documents are only 0.3 similar, they should ideally not hash to the same bucket, as they are not similar enough to be considered a match. The probability that two 30-percent similar documents will hash to the same bucket in any given band is 0.3^5, which is very low. However, across all 20 bands, there's still a 4.74-percent chance that these documents will become candidate pairs due to at least one band matching. These are considered false positives. While false positives do add to computational overhead by necessitating further checking, they are generally more acceptable than false negatives, but ideally, this number should also be less than 1-percent, particularly for computations involving large datasets where precision is crucial.

To summarize, the LSH algorithm aims to minimize false negatives and keep false positives within acceptable limits, ensuring that the computation remains efficient while reliably identifying documents that are similar above a certain threshold. The settings of bands and rows (b=20, r=5) are chosen to illustrate this balance, but in practice, these parameters might be tuned further to optimize the performance of the algorithm for specific use cases and document collection sizes.

LHS Trade-off Analysis:

A key concept in LSH is the tradeoff between the number of hash functions (rows of M) used to create signatures for each item and the way these signatures are divided into bands and rows, denoted by b and r, respectively. Adjusting these parameters affects the balance between false positives (dissimilar items incorrectly identified as similar) and false negatives (similar items not identified).

- Min-Hashes (Rows of M): This is the length of the signature for each item. The total number of hash functions used determines the resolution of the similarity measurement.
- Number of Bands (b): Each signature is split into b bands.
- Rows per Band (r): Each band contains r rows.

The Ideal Scenario:

In an ideal world, the similarity function between items would resemble a step function: if the actual similarity t is below a threshold s, items should never be considered similar; if t is above s, they should always be considered similar. However, this perfect separation is not achievable in practice without exhaustive comparison.

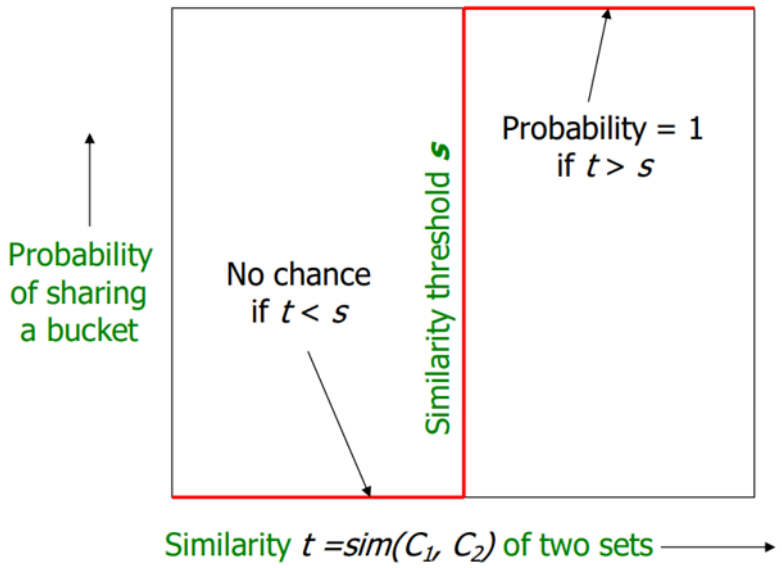


Figure 28: Ideal scenario

The Real-World Curve:

In reality, the probability of sharing a bucket (and thus being considered as candidate similar pairs) follows an S-curve rather than a step function. The shape of this curve can be manipulated by adjusting b and r. For any two items with similarity t, the probability that they match in a single band is t^r, and the probability that they do not match in any band is $(1-t^r)^b$. Consequently, the probability of at least one match across all bands is $1-(1-t^r)^b$.

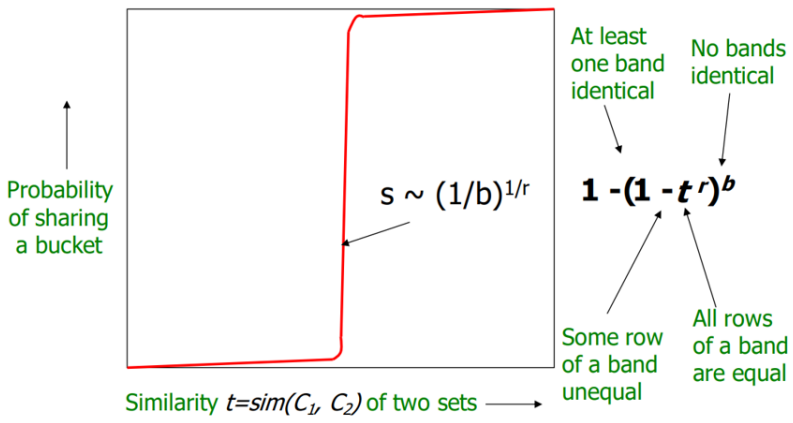


Figure 29: real-world scenario

Tuning the Parameters: The S-curve

The choice of b and r is influenced by the desired sharpness of the S-curve. A steeper S-curve near the similarity threshold s reduces the false negatives, while a broader curve increases false positives. The steepness can be increased by raising the number of bands b, tightening the region where items are likely to be considered similar.

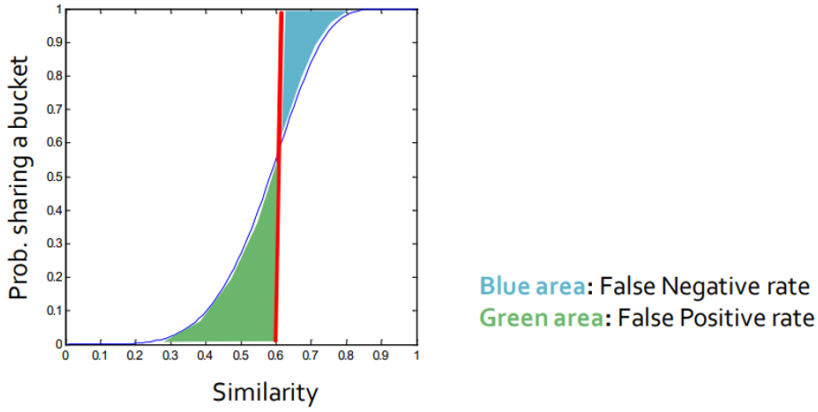


Figure 30: real-world scenario

Practical Considerations:

In practice, tuning b and r is not only an algorithmic decision but often a financial one, especially in large-scale applications such as web search engines. A higher number of bands might reduce the false negatives, improving the user experience by not presenting duplicate web pages. On the other

hand, it can significantly increase the computational resources required, as each additional band consumes more memory and processing power.

For instance, the Bing search engine uses LSH for duplicate detection in web crawling. The trade-off involves deciding how many computing resources can be dedicated to this task without incurring prohibitive costs. The goal is to minimize the green area in the S-curve graph, which represents false positives that require additional computation to eliminate, while also keeping the blue area, indicative of false negatives, as small as possible.

LSH Summary

The process of identifying similar documents in large datasets involves a sequence of hashing techniques, each serving a specific purpose in the data transformation and similarity detection pipeline.

1. **Shingling:** This initial step involves converting documents into sets of shingles, which are essentially the elemental components (like words or phrases) of the documents. Hashing is employed to map the verbose shingles into shorter, unique identifiers (IDs).

2. **Min-Hashing:** The large sets of shingles are then transformed into compressed representations known as min-hash signatures. These signatures preserve the similarity between the documents by ensuring that the probability of a match between the hashes of two shingles is equal to the similarity of the documents they come from.

3. **Locality-Sensitive Hashing (LSH):** In this final step, the min-hash signatures are divided into bands and rows. LSH aims to bucket these signatures in such a way that signatures (and therefore documents) with a high degree of similarity are likely to be hashed to the same buckets. Candidate pairs of signatures that are potentially similar are identified for further evaluation.

It is crucial to finely tune the parameters—number of bands (b), number of rows per band (r), and the matrix (M)—to capture most similar pairs while excluding dissimilar ones. An additional verification step in memory ensures that the candidate pairs do actually have similar signatures. Optionally, another pass through the data may be performed to confirm that the candidate pairs represent documents that are truly similar.

Mining Data Streams

Introduction to Data Streams in Big Data

In the ever-evolving landscape of Big Data, data streams represent an incessant flow of information that reflects the dynamic nature of the world around us. Unlike traditional datasets with a defined boundary, data streams are akin to a never-ending river of data points, where the origins are often unknown and the end is nowhere in sight. This chapter delves into the intricacies of mining data streams, a process critical to extracting meaningful insights from continuous, voluminous, and rapid data sources such as search engine queries, social media activity, and transaction records.

The realm of data streams challenges the conventional methodologies applied to static datasets. These streams are characterized by their infinite nature and non-stationary distribution, implying that assumptions regarding the data's behavior over time must be relinquished. The patterns,

trends, and relationships within a data stream are not only hidden but also constantly shifting, requiring sophisticated algorithms capable of adaptive learning and real-time processing.
As an illustration of the magnitude of data generated, consider the sheer volume of daily transactions processed by retail giants, the billions of searches conducted on search engines, and the hundreds of millions of social interactions captured online. These figures, while already staggering, continue to grow, underscoring the critical need for effective stream mining techniques to harness this wealth of information.
In the sections that follow, we will explore core algorithms designed to process and analyze data streams efficiently. From managing the stream's input rate to grappling with the data's unbounded nature, we embark on a journey to understand and implement strategies that can turn ceaseless data flows into actionable knowledge.

The Stream Model and Computational Challenges

Data streams are characterized by their rapid and continuous flow of elements, known as stream tuples, entering the system through various input ports. This relentless influx of data poses significant computational challenges, as it is infeasible for a system to store the entirety of the stream in an easily accessible manner. In practice, while some data may be archived for audit purposes or future analysis, such storage is often not conducive to real-time computation due to its format and accessibility limitations.

The central question then arises: *How can we perform critical calculations on a data stream using only a limited amount of memory?* This is not a trivial task, as the resources required to store and revisit the entire dataset are substantial and, in most cases, beyond our capacity. Real-time computations, such as tracking the frequency of likes per minute on social media or identifying trending topics, must be executed without the luxury of re-computing or extensively storing past data.

Time and Space Constraints

The constraints of time and space in stream processing are twofold. First, there is simply not enough memory to hold all incoming data. Second, the data cannot be stored and revisited, which rules out the use of algorithms that require multiple passes over the dataset. Traditional external memory algorithms, which are designed for handling datasets larger than the main memory, fall short as they do not support continuous queries and fail to deliver the requisite real-time responses.

As we venture deeper into the domain of data streams, these constraints guide our exploration of one-pass algorithms and innovative strategies tailored to operate within the confines of limited memory and time. The algorithms discussed in the following sections aim to tackle these challenges head-on, providing efficient and practical solutions for mining data streams in the context of Big Data.

Query Types in Data Streams

Interrogating data streams necessitates a bifurcated approach to querying: ad-hoc and standing queries. These two query classes serve distinct purposes and impose different requirements on the stream processing system.

Ad-hoc Queries

Ad-hoc queries are spontaneous, on-the-fly questions posed to the system, often in response to immediate business needs or curiosities. They are singular in nature, formulated and executed to answer a specific query at a given time. An example of an ad-hoc query might be: "*What is the maximum value seen so far in stream S?*" These queries are characterized by their unpredictability, as they are not known beforehand and can range widely in scope and complexity.

Standing Queries

In contrast, standing queries are persistent, ongoing queries that constantly monitor the stream for specific conditions or metrics. These are akin to a live dashboard, displaying real-time statistics or alerts about the incoming data. For instance, a network operator might use a standing query to continuously assess packet distribution to detect anomalies indicative of a cyber attack. An example of such a query could be: "*Report each new maximum value ever seen in stream S*".

Both types of queries are essential in a data stream environment. Standing queries allow for constant surveillance and quick detection of significant events, whereas ad-hoc queries provide the flexibility to extract specific information as needed. Together, they empower organizations to maintain vigilant oversight of their data streams while retaining the agility to delve into details on demand.

General Stream Processing Model

The general stream processing model represents a simplified yet robust architecture for handling continuous data streams. At its core lies a processor unit, responsible for executing queries on the incoming data streams. These streams, comprising a series of elements or tuples, enter the system at a relentless pace, necessitating efficient and rapid processing.

Processor Unit

The processor is designed to address two kinds of queries: standing and ad-hoc.

Storage Architecture

The underlying storage architecture within this model comprises two main components: archival storage and limited working storage. Archival storage serves as a vast repository—akin to a data lake in the lake house analogy—where all data streams are eventually stored. However, due to its sheer size and complexity, accessing and processing data directly from this storage is not practical for real-time computations.

The limited working storage, or the 'lake house,' contains a more manageable subset of data. This subset includes data tailored for the standing queries and any ad-hoc analysis. It's constrained in time but optimized for speed and accessibility, enabling the processor to perform real-time analytics effectively.

In essence, the general stream processing model captures the essential components of a data lake house architecture, balancing the depth of archival storage with the agility of limited working storage to cater to the ongoing needs of stream processing.

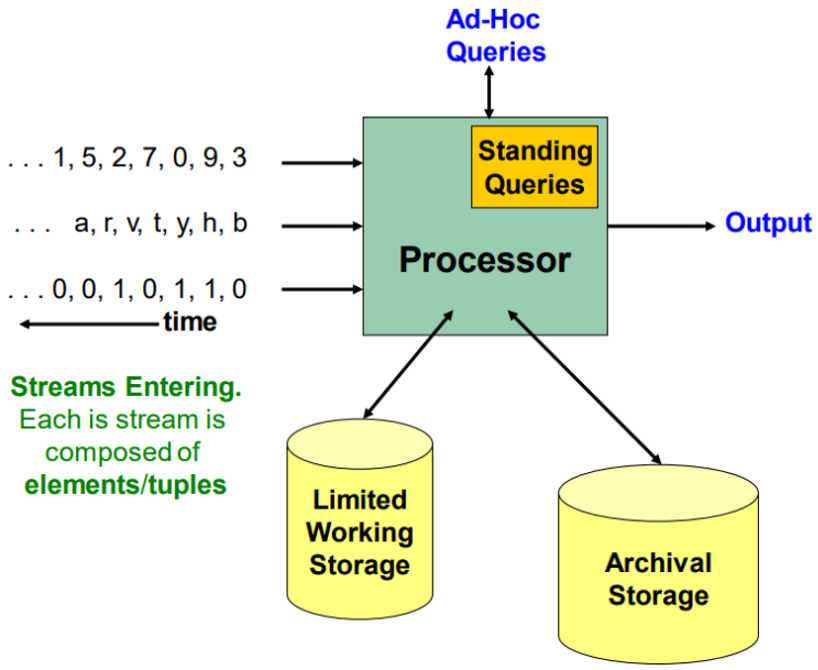


Figure 31: The General Stream Processing Model.

The model illustrated in the Figure encapsulates the flow of data streams into the system, the processing mechanisms for different types of queries, and the storage paradigms involved in managing large-scale data stream processing.

Applications of Mining Data Streams

Mining data streams is crucial across diverse domains, serving a multitude of applications that are imperative for operational insights and decision-making.

Query Stream Mining

Search engines, such as Bing, continuously analyze query streams to identify shifts in query frequencies. A surge in specific queries often signals real-world events, driving the need for dynamic adjustments in search algorithms and results prioritization. This analysis also extends to monitoring trends, identifying anomalies, and understanding user interests in real-time.

Click Stream Mining

Platforms like YouTube mine click streams to track video engagement, uncovering patterns in viewer behavior. A sudden spike in views for a particular video could indicate viral content, which warrants further investigation or moderation, depending on the nature of the video.

Social Network Feed Mining

Mining social network feeds, such as Twitter and Facebook, involves filtering through vast amounts of data to spot trending topics. This process often employs stream processing to handle the constant influx of user-generated content and to rapidly identify emerging discussions.

Network Traffic Monitoring

Monitoring IP packets at network switches is another critical application, where stream processing is used to detect unusual patterns that could indicate cybersecurity threats. A significant deviation from normal packet distribution might signify a denial-of-service attack, prompting immediate security measures.

Real-World Significance These applications illustrate the real-world significance of stream processing. Network security experts, for instance, liken a corporate network to a building with numerous doors: while all doors are expected to be secured, there are continuous attempts to test them. Detecting such attempts promptly is an ongoing battle, highlighting the necessity for robust stream processing capabilities in safeguarding digital assets.

Each of these applications underscores the importance of sophisticated stream processing solutions. They are not merely technical challenges but are central to the functionality and security of modern digital services and infrastructures.

Problems on Data Streams

The analysis of data streams entails various challenges and problem-solving scenarios, necessitating innovative algorithms to provide efficient solutions. We will focus on three representative problems that exemplify the unique aspects of stream processing.

Queries Over Sliding Windows

Sliding window queries are pivotal in data stream processing, where one seeks to count specific items within a continuously updating subset of the stream. The DGIM (Datar-Gionis-Indyk-Motwani) algorithm is particularly adept at counting the number of ones in a binary stream within the latest k elements, although its principles can be extended to broader counting tasks.

Filtering a Data Stream

Filtering a data stream is about selecting elements that satisfy a particular property, which is crucial for reducing the volume of data to be processed in real-time. Bloom filters offer a probabilistic approach to this problem, enabling quick determinations of whether an element with property x exists in the stream, without the need for full storage or exhaustive searching.

Counting Distinct Elements

The task of counting distinct elements within a window of the stream is a common problem, often solved by the Flajolet-Martin algorithm. This algorithm provides an approximation of the number of unique elements in the last k elements of the stream, a task that is otherwise computationally intensive given the stream's potential infinitude.

Sliding Window Model

A sliding window on a data stream offers a snapshot of the most recent items within a specified window length N. This model is particularly useful for processing streams of data in real-time, allowing for the observation and computation on the latest segment of the stream.

Window Length and Data Flow

Consider the window length N as the number of items from the stream that can be viewed or processed at any given time. As new data flows into the stream, the window slides forward, discarding the oldest item in favor of the newest. This ensures that computations are always based on the most recent data, which is often the most relevant for decision-making processes.

Practical Application

Companies like Amazon employ this model by maintaining a binary stream for each product. Each bit in the stream represents whether the corresponding product was part of a transaction, thus enabling the company to query the number of transactions involving a specific product within the last N transactions. Such information is critical for making business decisions and real-time recommendations.

Computing with a Sliding Window Computing the average of numbers in a sliding window is a straightforward process involving summing up the items within the window and dividing by N. When a new item enters the window, the oldest item is subtracted, the new item is added, and the average is adjusted accordingly. While this method requires maintaining the entire window in main memory, it exemplifies the simplicity of computing on a finite, manageable dataset.

Challenges Despite its utility, the sliding window model is not without challenges. The necessity to keep the entire window in main memory can be problematic, especially as N grows large. In subsequent sections, we will explore algorithms that circumvent this limitation by not requiring the entire window to be held in memory.

Counting Ones in a Sliding Window

When dealing with data streams, particularly in e-commerce platforms like Amazon, an essential query is determining the frequency of an event, such as the sale of a particular item. This leads us to the problem of counting ones in a binary stream within a sliding window.

Problem Definition

Given a binary stream of zeros and ones, we need an efficient method to answer queries about the number of ones in the last k bits of the stream, where $k \leq N$. This is equivalent to asking, for example, how many times a product has been sold in the last K transactions.

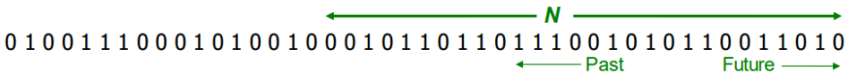


Figure 32: Illustration of counting ones within a sliding window of a binary stream.

Figure 32 depicts the concept of a sliding window moving over a binary stream, highlighting the need for algorithms that can efficiently count the number of ones without storing the entire sequence.

The DGIM Method for Counting Ones in a Sliding Window

Devising an algorithm that efficiently counts the number of '1's in a sliding window without the assumption of uniformity was accomplished by the introduction of the DGIM (Datar-Gionis-Indyk-Motwani) method. This algorithm is notable for not storing entire data streams but rather a compact summary that enables approximate counting.

Storing Bits with DGIM

The DGIM algorithm reduces the memory footprint by storing $O(\log^2 N)$ bits per stream, where N is the size of the sliding window. Despite its storage efficiency, DGIM ensures that the approximation is never off by more than 50%. For applications requiring higher precision, the method can be adjusted to store more bits, improving accuracy at the cost of additional space.

Timestamps in DGIM

Each bit in the stream is assigned a timestamp, and these timestamps are recorded modulo N to manage only the relevant ones within the current window. Since the timestamps are recorded modulo N, you need $log_2(N)$ bits to represent each timestamp (since $log_2(N)$ bits are sufficient to encode any number from 0 to N-1.) This technique allows representing any timestamp using $O(\log N)$ bits, significantly reducing the storage requirements.

Buckets in DGIM

DGIM introduces the concept of buckets—a segment of the window containing a record of two key pieces of information: (A) the timestamp of its end, and (B) the count of '1's (which are powers of two) between its beginning and end. The size of the bucket, in terms of the number of '1's it contains, is stored using $O(\log \log N)$ bits, leveraging the fact that only powers of two are counted. We can explain this complexity as follows:

- The largest bucket size is at most N, the size of the sliding window.
- Hence, the largest power of two that fits in the bucket is 2^k where $k = log_2(N)$.
- Since the size of the bucket is a power of 2, we don't need to store the actual number but only its power k.
- The exponent k can ragne from 0 to $log_2(N)$, hence to store k we need $log_(log_2(N))$ bits.

This constraint is central to the algorithm's storage efficiency and the 50% accuracy guarantee.

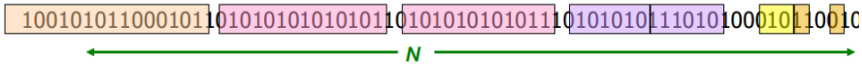


Figure 33: Illustration of the DGIM method for counting '1's in a sliding window.

The process described above and depicted in Figure 33 demonstrates how DGIM effectively balances the memory usage with the precision of the count, an essential requirement in big data applications such as monitoring transactions in an e-commerce setting.

Bucketization in the DGIM Method

The DGIM method employs a clever mechanism called bucketization to efficiently count the number of '1's in a binary stream. This process helps to store information about the stream in a compressed form while still providing a good approximation of the count of '1's.

Bucketized Stream

A bucketized stream is characterized by buckets, each representing a segment of the window and containing a count of '1's that are a power of two. These buckets are maintained under five crucial properties:

- Right side of the bucket should strictly strat from 1
- Length of bucket is equal to the number of 1's in it
- Each bucket size (the count of '1's it represents) is a power of two.
- At any time, there are at most two buckets for each size.
- Buckets do not overlap and are sorted by size in the internal representation. (Should not decrease bucket size as we move to the left).

Buckets that extend beyond the current window size, N, are discarded, as they no longer contain relevant information for the current window's count.

Updating Buckets

The updating process of buckets upon receiving a new bit is bifurcated into two scenarios:

1. If the new bit is '0', no update to the buckets is needed.
2. If the new bit is '1', a new bucket of size one is created. This triggers a possible cascading merge of buckets to maintain the power-of-two constraint. If there are now three buckets of a particular size, the two oldest are combined to form a bucket of the next size up, and this process continues recursively as necessary.

3. If the whole bucket is outside the sliding window, (i.e. if the current timestam minus the buckets timestamp is more than the sliding window size), then we no longer consider this bucket.

Each update may require scanning and combining buckets, with the number of operations bounded by $O(\log N)$ (since there are maximum of $log(N)$ buckets), ensuring that the method remains efficient even as the stream progresses.

Example of Bucketization Consider a binary stream where a new '1' arrives. Following the DGIM rules, a new bucket is created, and if this results in three buckets of the same size, the oldest two are merged. This process continues until the constraints are satisfied, and no more merges are required.

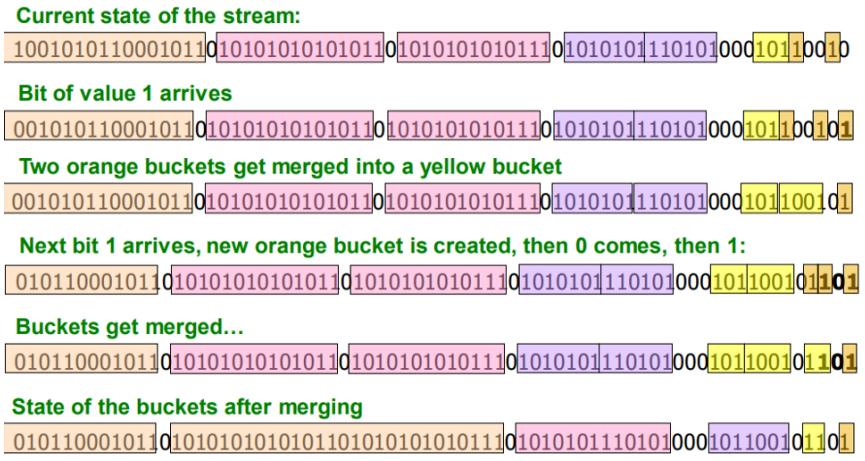


Figure 34: Example of a bucketized stream following the DGIM method.

Figure 34 visualizes the bucketized stream, showing the non-overlapping buckets sorted by size and illustrating how they evolve over time with the arrival of new bits.

Querying in the DGIM Method

Estimating the count of '1's within the most recent N bits is a crucial operation in the DGIM method. This estimation is accomplished with a simple yet effective querying process.

Estimating the Number of 1s

To estimate the number of '1's in the most recent N bits of the stream, the DGIM algorithm follows these steps:

1. Calculate the sum of the sizes of all buckets except the last. Here, "size" refers to the number of '1's each bucket represents.

2. Add half the size of the last bucket to the sum from step 1 to avoid overestimation, as we do not know precisely how many '1's from the last bucket fall within the desired window.

It is important to note that the algorithm does not provide the exact number of '1's, as we cannot determine how many '1's from the last bucket are still within the sliding window.

Example of DGIM Query

For instance, consider a bit-stream where we wish to know the count of '1's for $k = 10$. According to the DGIM rules, we would sum the sizes of all buckets and add half the size of the last bucket. If the result is an estimate of 6 but the correct answer is 5, the algorithm still falls within the acceptable margin of error, demonstrating its effectiveness in providing a rapid estimate.

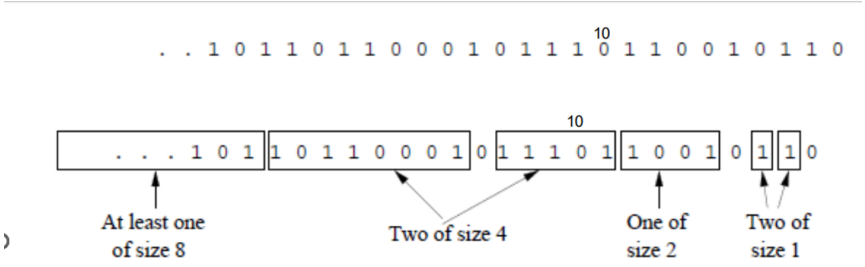


Figure 4.2: A bit-stream divided into buckets following the DGIM rules

- Number of 1s for k=10?
- Result (estimate): 6
- Result (correct): 5

Figure 35: An example demonstrating the querying process in the DGIM method.

Figure 35 shows the bit-stream divided into buckets as per DGIM rules, and the process of estimating the count for $k = 10$ using the summing and halving technique.

Error Bound space complexity and Extensions in the DGIM Method

Understanding the Error Bound

The error bound of the DGIM method is closely tied to the assumptions made about the last bucket in the sliding window. By design, the DGIM algorithm assumes that if the last bucket has size 2^r, only half of its '1's are within the window. This assumption can lead to an error of at most 2^{r-1}.

Given that there is always at least one bucket for each size smaller than 2^r, the actual sum of '1's is at least $2^r - 1$(this is the result of the geometric sum: 1+2+4+...). Therefore, the maximum error introduced by this assumption is 2^{r-1}, which is 50-percent of the value of the last bucket. Consequently, the algorithm guarantees that the error is at most 50-percent.

Reducing the Error

To reduce the error margin, the algorithm can be adapted to maintain more than two buckets for each size. Allowing $r-1$ or r buckets, instead of just 1 or 2, can reduce the error to at most $O(\frac{1}{r})$. This adjustment, however, increases the number of bits required to store the bucket information, establishing a trade-off between error precision and storage cost.

Extensions for Variable Query Sizes

While the basic DGIM algorithm is designed for querying the most recent N bits, it can be extended to handle queries for the last k bits where $k < N$. To estimate the number of '1's for such queries, one would find the earliest bucket that overlaps with k and sum the sizes of more recent buckets, adding half the size of the overlapping bucket. This extension employs the same principles as the standard DGIM querying mechanism.

Understanding the space complexity:

To understand the space complexity of the DGIM algorithm, we begin by stating the relevant facts about its components and operations:

- **Timestamp Storage:** Each bit in the stream is assigned a timestamp modulo N, requiring $O(\log N)$ bits.
- **Bucket Storage:** Each bucket stores:
  - The timestamp of its most recent '1', which requires $O(\log N)$ bits.
  - The size of the bucket, which is a power of two. Representing the size as an exponent requires $O(\log \log N)$ bits.
- **Bucket Updates:** Updating buckets (insertion and merging) may require scanning and combining buckets, with the number of operations bounded by $O(\log N)$.

Given the above facts, we can derive the overall space complexity of the DGIM algorithm as follows:

- **Number of Buckets:** At any given time, the number of buckets is $O(\log N)$. This is because the bucket sizes are powers of two, and the maximum bucket size that can be present in the window is $2^{\log_2 N} = N$, leading to at most $\log_2 N$ different bucket sizes.
- **Space Per Bucket:**
  - Each bucket requires $O(\log N)$ bits to store its timestamp.
  - Each bucket requires $O(\log \log N)$ bits to store its size.
  - Hence, the total space required for each bucket is $O(\log N) + O(\log \log N) = O(\log N)$ bits.
- **Total Space for All Buckets:**

– Since there are $O(\log N)$ buckets, and each bucket requires $O(\log N)$ bits of storage, the total space required for storing all buckets is:

$$O(\log N) \times O(\log N) = O(\log^2 N).$$

Therefore, the overall space complexity of the DGIM algorithm is $O(\log^2 N)$. This ensures that the algorithm remains efficient in terms of storage, even as the size of the sliding window N increases. This sublinear space complexity makes the DGIM algorithm highly suitable for processing large-scale data streams.

Filtering Data Streams

Filtering is a fundamental operation in data stream processing, where the goal is to determine which elements of the stream match a set of given criteria.

Challenges in Filtering

While a hash table is the conventional approach to determine if an element of a data stream matches a given set of keys, there are practical limitations. Specifically, the size of the hash table can become a constraint when dealing with massive sets of keys, as there might not be enough memory available to store the entire set.

Applications Email spam filtering and web crawling are quintessential examples of applications that require efficient data stream filtering. For instance, in email spam filtering, having a list of "good" email addresses can help quickly filter out non-spam emails without expensive content analysis. In web crawling, a list of already crawled URLs needs to be maintained to avoid redundant work.

Bloom Filters: A Solution

Bloom filters provide an efficient alternative to hash tables for filtering data streams. They are a type of probabilistic data structure that allow for quick membership tests with a compact memory footprint. Bloom filters work by using multiple hash functions to map elements to a bit array.

How Bloom Filters Work When an element is added to a Bloom filter, it is hashed by several different hash functions, and each hash function sets a bit in a fixed-size bit array. To test if an element is in the set, the element is hashed using the same hash functions, and the corresponding bits in the array are checked. If all the bits are set, the element is assumed to be in the set; if any bit is not set, the element is definitely not in the set.

- **Definition:** Let S be the set with $|S| = m$ elements, and B a bit array with $|B| = n$ bits. Bloom Filters employ k independent hash functions $h_1, h_2, \ldots, h_k$.
- **Initialization:**
  1. Initialize bit array B to all zeros.
  2. For each element s in S, hash s with each hash function h_i and set $B[h_i(s)] = 1$ for $i = 1, \ldots, k$.

- **Run:**
    1. For an incoming stream element with key x, compute $h_i(x)$ for all $i = 1, \ldots, k$.
    2. If $B[h_i(x)] = 1$ for all i, the element x is likely in S.
    3. If any $B[h_i(x)]$ is 0, then x is definitely not in S.

Advantages and Trade-offs The main advantage of Bloom filters is their space efficiency. They require significantly less memory compared to hash tables, especially when handling large sets. However, Bloom filters can produce false positives, meaning they can incorrectly indicate that an element is in the set when it is not. Despite this, they never produce false negatives, ensuring that all elements reported as not in the set are indeed not present.

Use Cases Due to their efficiency, Bloom filters are widely used in applications where memory is constrained and quick membership tests are essential. In email spam filtering, Bloom filters can efficiently store and check large lists of "good" email addresses. In web crawling, they help manage large sets of already crawled URLs, reducing redundant requests and saving bandwidth.

By leveraging Bloom filters, systems can achieve efficient data stream filtering with minimal memory usage, making them an ideal solution for high-throughput environments where conventional data structures fall short.

Bloom Filter Analysis

To understand the effectiveness of Bloom Filters, it is essential to analyze the probability of false positives, which is crucial for gauging their reliability.

- Let $|S| = m$ be the number of elements to filter, $|B| = n$ the size of the bit array, and k the number of independent hash functions used.
- The fraction of the bit vector B that is expected to be set to 1s after all insertions is $(1 - e^{-km/n})$, assuming a uniform distribution.
- The false positive probability, when checking all k hash functions, is given by $(1 - e^{-km/n})^k$.

The analysis further provides insight into the optimal number of hash functions to use in order to minimize the false positive probability. For a set S with m elements and a bit array B with n bits:

1. With one hash function ($k = 1$), the false positive probability is $(1 - e^{-1/8}) = 0.1175$.
2. Increasing to two hash functions ($k = 2$), the probability drops to $(1 - e^{-1/4})^2 = 0.0493$.
3. As k increases, there is an "optimal" number of hash functions given by $n/m \ln(2)$, which minimizes the false positive rate.
4. In our example with $m = 1$ billion and $n = 8$ billion, the optimal $k \approx 5.54$, which we round up to 6. At $k = 6$, the false positive rate is $(1 - e^{-1/6})^6 = 0.0235$.

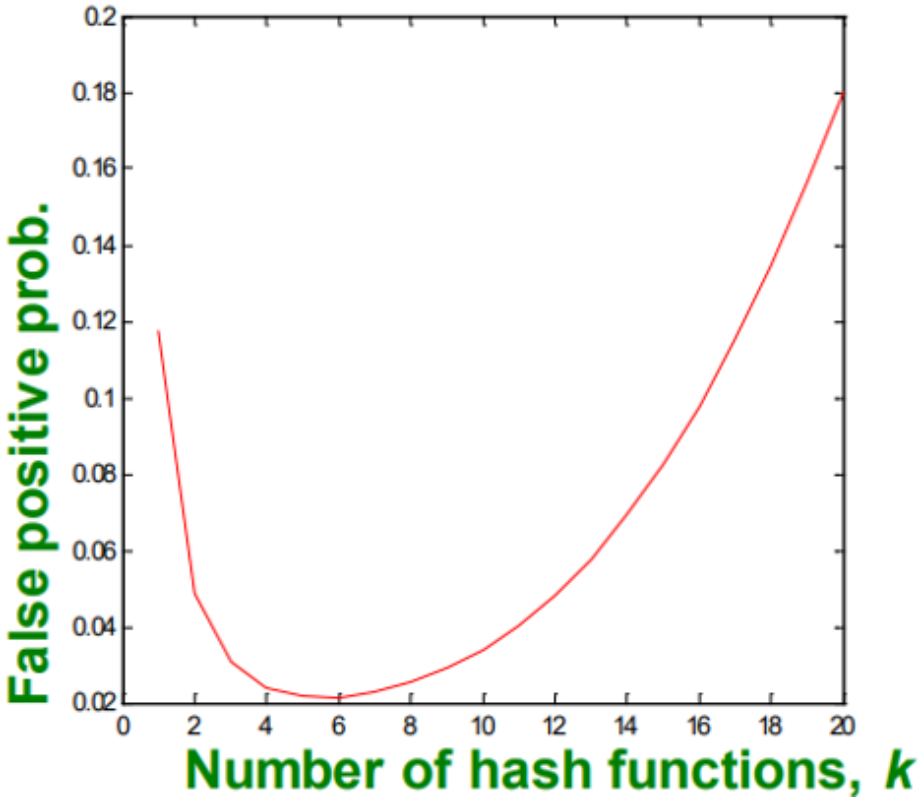


Figure 36: The relationship between the number of hash functions and the false positive probability in a Bloom Filter.

The false positive probability is significantly reduced by employing multiple hash functions, and the optimal number of hash functions can be calculated given the sizes of the set S and the bit array B. This allows us to efficiently tune the Bloom Filter for a balance between space complexity and error rate.

Bloom Filter Conclusion

Bloom Filters are a powerful tool in the field of data processing and have a wide range of applications due to their space efficiency and computational simplicity.

- **Guaranteed No False Negatives:** By design, Bloom filters never return a false negative, making them reliable for ensuring an element is not mistakenly excluded if it is part of the set.
- **Memory Efficiency:** They utilize a compact bit array to represent a set, significantly saving memory, especially when dealing with large datasets.
- **Pre-processing:** Bloom filters are ideal for pre-processing steps where they can quickly filter out non-matching elements before applying more resource-intensive checks.

- **Hardware Suitability:** Due to their simplicity, Bloom filter operations, especially hash computations, can be parallelized and are well-suited for hardware implementation.
- **Dynamic Nature:** While adding elements to a Bloom filter is straightforward, removing elements is not as trivial due to potential hash collisions. Thus, Bloom filters are generally static, built once for a given set of elements.
- **Practical Application:** They are used in various domains, such as network security for MAC filtering, search engines for lossy index structures, and many more areas where a trade-off between accuracy and efficiency is acceptable.
- **Research and Variations:** The field of probabilistic filters continues to be a rich area of research, aiming to optimize the trade-off between computational overhead, memory usage, and false positive rates.

In conclusion, the utility of Bloom filters in scenarios that can tolerate a controlled false positive rate while requiring quick membership checks makes them an indispensable tool in big data and streaming analysis applications.

Counting Distinct Elements in Data Streams

When analyzing data streams, a common problem is to maintain an accurate count of the number of distinct elements that appear over time. This section describes the problem and some typical applications.

Problem Statement

- A data stream consists of elements chosen from a universe of size N.
- The goal is to maintain a count of the distinct elements observed so far in the stream.

Naïve Approach

The straightforward approach is to maintain a hash table to keep track of all distinct elements that appear in the stream. However, this method is not space-efficient, as the number of distinct elements can be very large.

Applications

- **Web Crawling:** Determining the number of distinct words found on web pages can help identify potential spam by detecting unusually low or high uniqueness in content.
- **Online Services:** Counting unique users or actions, such as the number of unique visitors to a social media platform or distinct web pages requested by a customer, is vital for understanding user engagement.
- **Commerce:** Keeping track of the number of distinct products sold in a given time frame is important for inventory management and sales analysis.

Using Small Storage

- The real challenge arises when there is not enough space to maintain the set of all distinct elements.
- An acceptable solution is to estimate the count, allowing for some error, while limiting the probability that this error is large.
- Approaches such as probabilistic data structures, which will be discussed, offer a way to get a reasonable estimate of the count without the need for large storage.

Remark: The lecturer emphasizes the importance of hashing functions in this context, indicating that while exact counts are ideal, the constraints of space make it necessary to find efficient algorithms that can approximate counts with minimal error.

The Flajolet-Martin

The Flajolet-Martin algorithm is a probabilistic technique used for approximating the number of distinct elements in a data stream. It is based on hashing elements and analyzing the pattern of zeros in the hash values.

Hash Function

- A hash function h is selected such that it maps each element to at least $\log_2 N$ bits, where N is the size of the universe of elements.

Trailing Zeros

- For each element a in the stream, let $r(a)$ denote the number of trailing zeros in the binary representation of $h(a)$.
- The position of the first 1 bit, counting from the right, is recorded. For example, if $h(a) = 12$, then in binary $h(a) = 1100$, and $r(a) = 2$.

Estimation

- Keep track of the maximum $r(a)$ seen so far, denoted as R.
- The estimated number of distinct elements in the stream is 2^R.

Example

An illustrative example for this algorithm is given with the input stream $a = 1, 2, 1, 2, 3, 4, 3, 1, 2, 3, 1$ and a simple hash function $h(x) = 6x + 1 \mod 5$. Processing this stream, the maximum $r(a)$ observed is 2, thus estimating the number of distinct elements to be $2^R = 2^2 = 4$, which coincides with the true number of distinct elements in this particular example. See image below:

- Input stream =1,3,2,1,2,3,4,3,1,2,3,1
- h(x)=6x + 1 mod 5

h(1)=2=010 r=1
h(3)=4=100 r=2
h(2)=3=011 r=0
h(1)=2=010 r=1
h(2)=3=011 r=0
h(3)=4=100 r=2
h(4)=0=000 r=0
h(3)=4=100 r=2
h(1)=2=010 r=1
h(2)=3=011 r=0
h(3)=4=100 r=2
h(1)=2=010 r=1

- $R = \max_a r(a)=2$
- Distinct elements (estimated): $2^R = 2^2 = 4$

Understanding the Flajolet-Martin Algorithm

The key to the Flajolet-Martin algorithm's success lies in its use of hash functions and the properties of binary representations:

- If a hash function uniformly distributes elements across binary outputs, about $\frac{1}{2}$ of the hashed elements will end with a zero, about $\frac{1}{4}$ will end with two zeros, and so on.
- If the maximum number of trailing zeros observed in the hash values is R, it implies that roughly 2^R different elements have been seen, because 2^R is the expected number of elements to see before finding one that hashes to a value ending in R zeros.
- The algorithm is robust against repetitions, as the hash of a repeated element will not affect the maximum R value.

Increasing Precision

To enhance precision, the algorithm can be repeated with multiple hash functions:

- A simple average of estimates from multiple hash functions can be distorted by extreme values.

- A better approach is to take the median of the estimates to mitigate the impact of outliers.
- Further refining the estimate involves grouping hash functions, taking the median within each group, and then averaging these medians across groups.

Remark: The probabilistic nature of the algorithm means it provides an estimate rather than an exact count. The accuracy of this estimate can be influenced by the distribution of input data and the properties of the chosen hash functions.

Adwords

The Adwords Model

The Adwords Model is a structured approach used in online advertising to manage and display advertisements in search engine results. The model is based on several key components:

- A set of bids by advertisers for search queries.
- A click-through rate for each advertiser-query pair.
- A budget for each advertiser, typically set for a defined period, such as one month.
- A limit on the number of ads to be displayed with each search query.

When responding to a search query, the Adwords system selects a set of advertisers according to the following criteria:

- The size of the set of advertisers is no larger than the limit on the number of ads per query.
- Each advertiser has bid on the search query.
- Each advertiser has sufficient budget remaining to pay for the ad if it is clicked upon.

This ensures that the ads displayed are relevant, within budget, and capped at a maximum number for clarity and quality of user experience.

Simplification and Matching in the Adwords Model

The Adwords model can be simplified for a clearer understanding of its mechanisms. The simplification assumes the following conditions:

- Each user has only one advertising slot. If multiple slots are available, the ad selection process is repeated for each slot.
- Every advertisement is assumed to have the same expected revenue.

This approach leads us to a specific class of algorithms. Visually, this can be represented as a timeline with users and their search queries on one side, and on the other side, the businesses. Each business is associated with a budget (denoted in dollars) and the keywords they are bidding for.

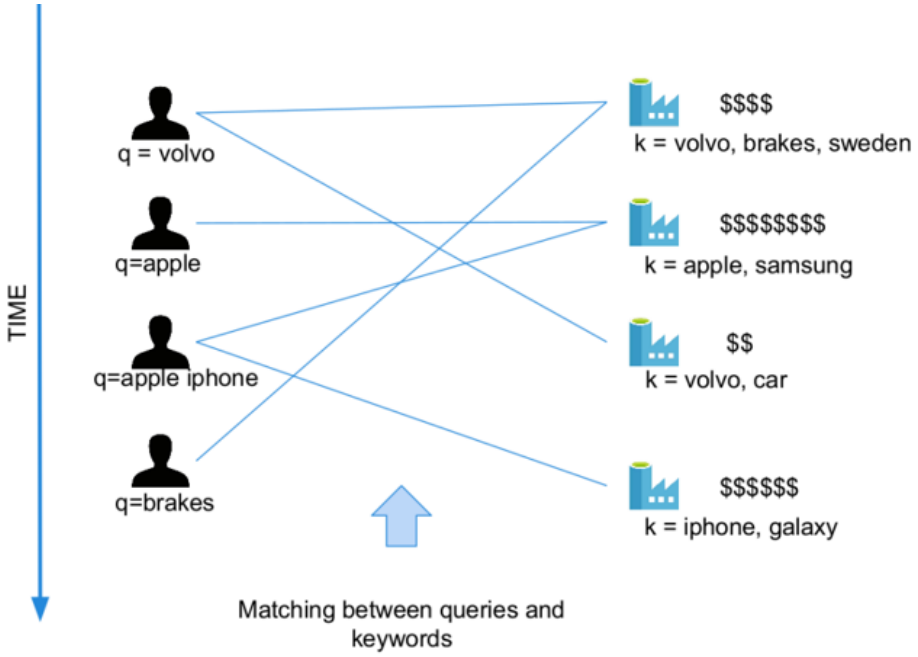


Figure 37: Matching between queries and advertisers

This scenario sets up a matching problem:

1. As queries come in, the system must match them with appropriate advertisers.
2. For example, a query for "Volvo" should be matched with an advertiser who has bid for that keyword.

Such a model emphasizes the importance of efficiently pairing the incoming search queries with the best-suited advertisements, taking into account the constraints of advertisement slots and expected revenue.

Detour: Bipartite Matching

Bipartite matching is a foundational concept in graph theory, essential for understanding many algorithms, including those used in the Adwords model. A bipartite graph consists of two sets of vertices where every edge connects a vertex from one set to a vertex from the other set. In the context of an analogy with boys and girls, matchings in a bipartite graph can be seen as a form of pairing between the two distinct sets.

- A **matching** is a subset of edges such that no node is the endpoint of more than one edge.
- The **cardinality of a matching** is the number of edges in the matching.
- A **perfect matching** occurs when all vertices in the graph are matched, meaning no vertex is left unmatched.

- A **maximum matching** is a matching that contains the largest possible number of matches.

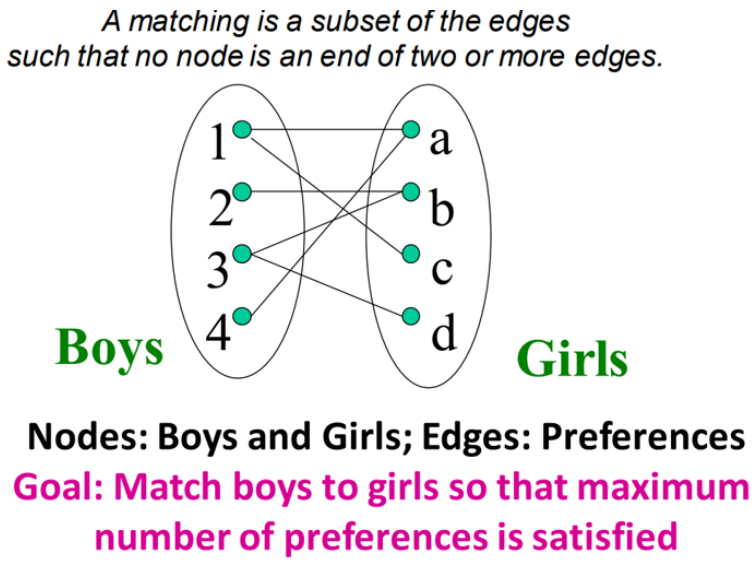


Figure 38: An example of a bipartite graph with a possible matching.

This graphical representation is useful for visualizing the matchmaking process, albeit in a simplified and, as noted, somewhat "lame" fashion that excludes other types of pairings for the sake of this example.

Understanding Matchings in Bipartite Graphs

In a bipartite graph, a matching can be demonstrated as a set of paired connections that do not overlap. For example, if we consider the graph vertices to represent boys and girls, a matching would be a set of pairs where each boy is connected to a girl, with no individual being part of two pairs simultaneously.

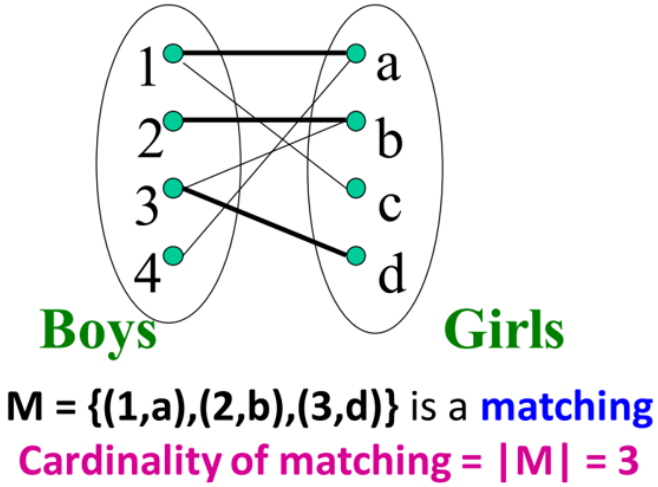


Figure 39: An example of a bipartite matching with no overlapping connections.

Perfect and Maximum Matchings

A perfect matching is achieved when every vertex in the graph is included in the matching, ensuring that no vertex remains alone. This is also referred to as a maximum matching when considering all possible combinations of matchings, and it represents the largest set of pairs that can be formed.

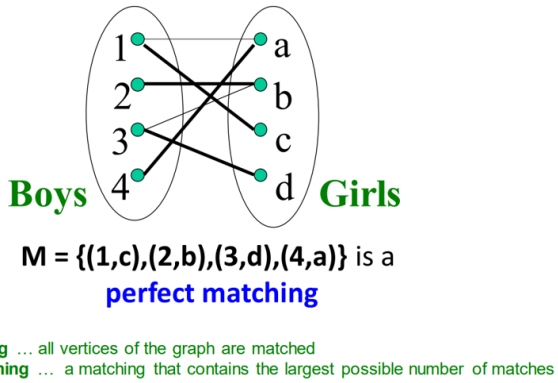


Figure 40: A bipartite graph showcasing a perfect matching.

Such concepts are central not only to the analysis of algorithms but also to their practical application in systems like the Adwords model, where the goal is to find an optimal matching between queries and advertisements.

Online and Offline Algorithms

When discussing algorithms, especially in the context of Big Data and Adwords, it is important to distinguish between **offline** and **online** algorithms.

Offline Algorithms

The classic model of algorithms is referred to as *offline algorithms*. In this model:

- The entire input data is available from the beginning.
- Computations can be performed on the full data set, such as sorting or applying functions.

Offline algorithms provide a full overview of the data, allowing for comprehensive data processing.

Online Algorithms

In contrast, *online algorithms* operate under a different set of constraints:

- Input data is received sequentially, one piece at a time.
- Decisions must be made immediately and are irrevocable—once a decision is made, it cannot be undone.

This is similar to the data stream model and is particularly relevant for dynamic systems like Adwords. In such a system, you receive queries one at a time and must respond without knowing future events. While predictions can be made based on historical data, the exact nature of future queries is unknown. This necessitates the use of online algorithms, where each query must be treated as it comes, making them ideal for handling real-time data and immediate decision-making required in online advertising.

The Greedy Algorithm in Adwords

The **Greedy Algorithm** is an intuitive approach used in the Adwords model, which, despite its simplicity, underpins the operations of this substantial online advertising market.

Greedy Algorithm for Query Selection

For each incoming query, the algorithm operates as follows:

- Selects "randomly" an advertiser who has bid for that specific query.

This method aligns with the principle of a greedy algorithm, which consistently makes the locally optimal choice at each stage with the hope of finding the global optimum.

Maximal Matching vs Greedy Algorithms

The Adwords problem can be analogized to finding a maximal matching in a bipartite graph, where the goal is to match queries to advertisers while maximizing the overall effectiveness of these pairings.

- The greedy algorithm aims for a solution that may not necessarily be perfect but is sufficient for the Adwords model.
- It represents a class of algorithms that are opportunistic, taking what appears to be the best immediate option.

Despite its name, the greedy algorithm is not short-sighted. In certain scenarios, like the Adwords model, it effectively balances between achieving a satisfactory matching quickly and managing computational complexity.

Competitiveness and the Competitive Ratio

An essential concept to understand when evaluating the performance of algorithms, is **competitiveness**.

Defining the Competitive Ratio

The competitive ratio is a measure used to evaluate an algorithm's performance relative to the optimal solution:

- Let I represent the input, M_{greedy} the matching produced by the greedy algorithm, and M_{opt} the optimal matching.
- The competitive ratio (ρ) is defined as the worst-case ratio of the greedy algorithm's matching size to the optimal matching's size over all possible inputs.

$$\rho = \min_{\text{all possible inputs } I} \left(\frac{|M_{\text{greedy}}|}{|M_{\text{opt}}|} \right)$$

This ratio provides insight into the greedy algorithm's worst performance compared to an optimal solution across all possible scenarios.

Interpreting the Competitive Ratio

Through analysis, one might find that the competitive ratio for a given greedy algorithm is $\frac{1}{2}$ or 0.5, indicating that, in the worst case, the algorithm performs at half the effectiveness of the optimal solution.

- This would mean that, for the worst-case input, the greedy algorithm can only match half as many queries to advertisers as the optimal matching could.
- Such a competitive ratio may not seem high, but it's crucial to consider the context of the problem and the computational resources required for more complex algorithms.

The consideration of the competitive ratio allows for a better understanding of the algorithm's efficiency and helps determine the balance between computational complexity and the quality of the solution provided.

The Balance Algorithm

In the pursuit of more effective matchings in the Adwords model, the **Balance Algorithm** emerges as a significant improvement over the greedy approach.

Concept and Mechanism

The Balance Algorithm enhances the match rate and, consequently, the potential revenue by adjusting the selection process:

- Instead of randomly selecting an advertiser, it chooses the advertiser with the largest remaining budget for the query.

Advantages of the Balance Algorithm

The incentives for using a more advanced algorithm like the Balance Algorithm include not only increased revenue but also improved user quality:

- Advertisers with more budget are typically more invested and possibly offer more legitimate deals, reducing the presence of low-quality or scam advertisements.
- A larger budget indicates the advertiser's commitment and suggests that their advertisements are likely to be more relevant and less fraudulent.

By selecting advertisers based on their remaining budget, the algorithm also implicitly aligns with Google's interest in maintaining high-quality ads and ensuring a positive user experience. The improvement over the greedy algorithm lies in its ability to sustain ad quality while maximizing revenue, thereby balancing both advertiser and platform interests.

Competitiveness of the Balance Algorithm

The effectiveness of an online algorithm can be quantitatively assessed by its competitive ratio, and the Balance Algorithm stands out in this aspect.

Competitive Ratio of Balance

For the Balance Algorithm used in Adwords:

- In the general case with numerous advertisers, the competitive ratio is $1 - \frac{1}{e}$, which approximates to 0.63.
- This ratio has been proven through mathematical derivations involving Taylor series and the Euler expansion.

Optimality Among Online Algorithms

An interesting note about the Balance Algorithm is that:

- No other online algorithm has been shown to surpass this competitive ratio for the Adwords problem.
- Proofs exist, referenced in academic texts and sources like Wikipedia, that substantiate this claim.

This high competitive ratio indicates a significant degree of optimality, suggesting that the Balance Algorithm manages to achieve near-optimal results in a dynamic, real-time environment, and there is no known online algorithm that can consistently perform better in the context of Adwords.

Limitations of the Balance Algorithm: A Failure Example

While the Balance Algorithm is generally effective, there are scenarios where its approach can lead to suboptimal results.

Illustration of a Balance Failure

Consider the following case with two advertisers bidding for the keyword "volvo":

- Advertiser A has a budget of $55 and bids $1 per click.
- Advertiser B has a budget of $50 and bids $10 per click.

Using the Balance Algorithm's criterion of selecting the advertiser with the largest remaining budget could result in consistently choosing Advertiser A, due to their larger budget. However, this leads to less revenue than selecting Advertiser B, who bids more per click.

- Balance outcome: 5 clicks at $1 each, totaling $5.
- Optimal alternative: 5 clicks at $10 each, totaling $50.

Competitive Ratio Implications

In such cases, the competitive ratio of the Balance Algorithm can effectively drop to zero, revealing its vulnerability:

- The competitive ratio, in this case, reflects the algorithm's worst-case performance, and with this setup, it demonstrates a significant failure.
- If advertisers were to strategize around this algorithm's predictability, they could exploit it to diminish its effectiveness.

This example underscores the necessity for continual improvement of the Balance Algorithm to mitigate such pitfalls and ensure it remains robust against strategic bidding.

The Generalized Balance Algorithm

The Generalized Balance Algorithm refines the approach to selecting ads, aiming to address the limitations of the standard Balance Algorithm.

Adjusting the Selection Criteria

This enhanced algorithm introduces a bias towards ads with higher bids while being more nuanced about the remaining budget:

- The algorithm considers the fraction of the budget remaining for each advertiser, allowing for a proportional selection that aims to utilize parts of each advertiser's budget.

Dealing with Arbitrary Bids

For a given query q and bidder i, the algorithm assesses:

- The bid x_i
- The total budget b_i
- The amount spent so far m_i
- The fraction of the budget left $f_i = 1 - (m_i/b_i)$
- It calculates a value $\psi_i(q) = x_i(1 - e^{-f_i})$

Allocation Strategy

The ad selection is then determined by:

- Allocating query q to the bidder i with the largest value of $\psi_i(q)$.

Competitive Ratio

The Generalized Balance Algorithm maintains:

- The same competitive ratio of $1 - 1/e$ as the original Balance Algorithm.

By considering both the bid amount and the proportional remaining budget, the Generalized Balance Algorithm provides a sophisticated method for ad allocation that strives to maximize revenue and maintain the quality of displayed ads.

Real-World Considerations in Ad Matching

Real-life scenarios present challenges that algorithms like Adwords must contend with to be truly effective.

Variations in Budgets and Bids

In practice, companies have vastly different budgets and their bids can vary greatly:

- Budgets can range significantly, impacting the strategy for bid placement.
- Bids for keywords are not uniform; they differ based on the value companies place on each keyword.

Dynamics of the Bidding Landscape

The landscape of bidders is in constant flux:

- New companies enter the auction, while others may leave or adjust their campaign strategies.
- The number of keywords bid on is massive, often reaching millions, necessitating automated systems to manage them.

Beyond Simple Keyword Matching

Keyword matching involves more than just exact terms:

- Broad matching is commonly used to encompass variations and misspellings of keywords.
- The matching process is sophisticated, often leveraging user behavior and other signals to optimize ad placement.

Privacy and Ethical Considerations

With the precision of modern ad targeting, privacy and ethical concerns become paramount:

- Advertisers can target specific demographics with incredible accuracy, raising questions about privacy and the appropriate use of user data.
- Balancing effective ad targeting with societal norms and individual privacy rights is an ongoing challenge in the industry.

Understanding these real-life complexities is crucial for developing algorithms that are not only technically proficient but also ethically and socially responsible.

Recommender Systems: Content-based Systems and Collaborative Filtering

Introduction

Recommender Systems (RS) are an integral part of many modern platforms, guiding users towards content and products by predicting preferences. This section explores the mechanisms behind these systems, including content-based and collaborative filtering methods.

Goals

- Understand the motivation behind the development of recommender systems (RS).
- Grasp how RS operates and learn about the principles behind various RS methods and algorithms.

Motivation

The impetus for RS comes from the need to navigate the vast array of options available online, providing users with relevant suggestions. This, in turn, benefits content providers and websites by driving engagement and satisfaction. The challenge lies in creating a system that harmonizes the interests of all stakeholders: users, content partners, and the platforms themselves.

Recommendations in Recommender Systems

Recommender systems bridge the gap between user needs and the vast amount of options available. They guide users in discovering items—such as products, websites, or media—that align with their preferences and behaviors. Major platforms like Amazon, Netflix, Facebook, Spotify, and YouTube utilize these systems to personalize the user experience, increasing both user satisfaction and platform engagement.

Figure 41: Interaction between user search and recommendations. Examples of platforms using recommender systems.

In the visual depiction, the user is at the center, contemplating their choices. The recommendation system acts as a conduit between their search queries and the pool of available items, creating

a dynamic and responsive ecosystem that caters to both explicit searches and suggested recommendations. This two-way interaction is the cornerstone of a personalized user experience, making recommendations an essential component of modern digital platforms.

From Scarcity to Abundance

The digital age has transformed the way information about products is disseminated. Historically, shelf space in traditional retail was limited, as was the case with TV networks and movie theaters. This scarcity required selective curation and limited what could be offered to consumers.

With the advent of the web, the constraints of physical space have been virtually eliminated, leading to near-zero-cost dissemination of information about products. This transition from scarcity to abundance has given consumers a deluge of choices.

The consequence of this shift is clear: more choices necessitate better filters. Recommender systems have become the modern solution to this problem, effectively curating personalized experiences for users and aiding in the discovery of products that may otherwise remain obscure.

The capacity to manage and present this abundance of choices effectively is not just a technical challenge but a crucial business strategy that recommender systems strive to address.

The Long Tail Phenomenon

The Long Tail phenomenon, as it relates to Recommender Systems, refers to the strategy of selling a large number of unique items with relatively small quantities sold of each (the "long tail") as opposed to selling only a small number of best-selling items (the "head"). This concept has gained prominence with the advent of digital marketplaces and content platforms.

- **Head**: This part of the curve represents a small number of items that are high-impact, popular, mainstream, and with high visibility and sales.
- **Long Tail**: Contrary to the head, the long tail represents a large number of items that are low-impact, niche, and obscure, yet collectively they represent a significant portion of the market.

Recommender systems play a crucial role in surfacing long tail items to users, allowing even less popular products to find their audience. This expands consumer choice and can lead to greater customer satisfaction as users find products that more closely match their tastes and needs.

Types of Recommendations

Recommender systems can generate various types of recommendations to cater to different needs and contexts. We categorize them as follows:

- **Editorial and Hand Curated:** These are recommendations based on a curated list of favorites or essential items. These lists are usually created by experts or through editorial oversight and aim to guide users towards high-quality or important content.
- **Simple Aggregates:** This type includes recommendations that are generated based on aggregate data such as the 'Top 10' lists, 'Most Popular' items, or 'Recent Uploads'. These are typically based on simple metrics that reflect user engagement or recentness.

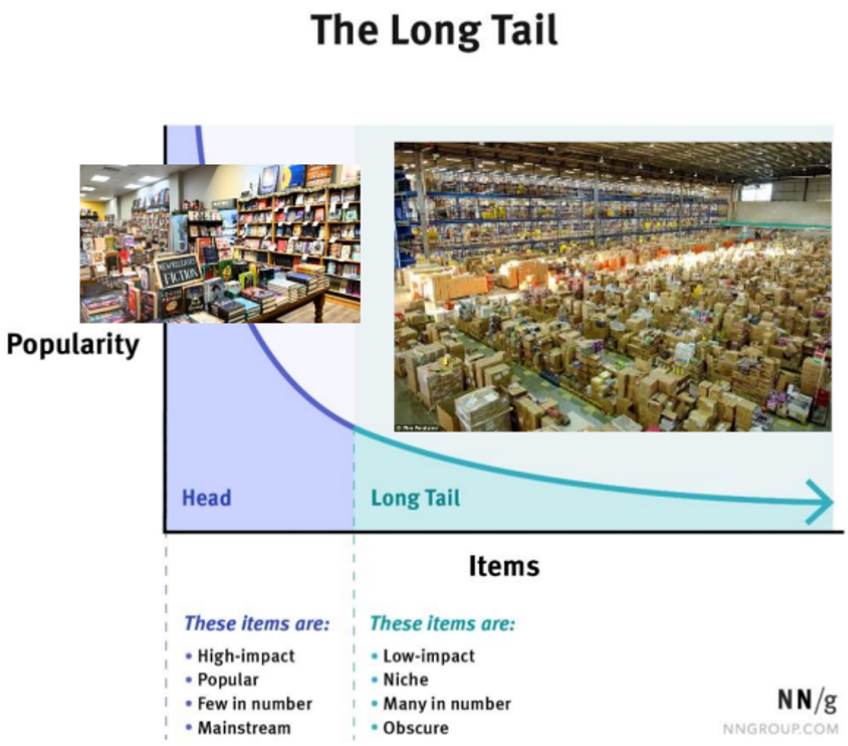


Figure 42: The Long Tail graph illustrating popularity versus items, showing both the 'head' of high-impact, popular, and mainstream items, and the 'long tail' of low-impact, niche, and obscure items.

- **Tailored to Individual Users:** Personalized recommendations are tailored to the individual preferences and behaviors of each user. This is the most sophisticated type of recommendation, used by platforms like Amazon, Netflix, Apple Music, Spotify, and even in targeted advertising like Facebook Ads. These systems utilize complex algorithms to predict what a user may like based on their past interactions.

Each type of recommendation serves a different purpose and is used by different platforms depending on their goals and the user experience they wish to provide.

Formal Model of Recommender Systems

A formal model of a recommender system is foundational to understanding how recommendations are generated. In this model, we define the following:

- X represents the set of Customers who interact with the system.
- S denotes the set of Items that the system can recommend.

- The Utility function $u : X \times S \to R$ is a function that assigns a rating to an item for a customer.

The ratings set R is a totally ordered set representing the possible ratings that an item can receive, such as a score from 0 to 5 stars, or a real number within the range [0,1].

This model serves as the abstract representation of the processes within a recommender system, allowing for the application of various algorithms to predict user preferences and suggest relevant items.

$$
\begin{aligned}
X &= \{\text{set of Customers}\} \\
S &= \{\text{set of Items}\} \\
R &= \{\text{set of ratings}\} \\
u &: X \times S \to R
\end{aligned}
$$

Utility Matrix

A utility matrix is a fundamental component of many recommender systems. It is a structured way to represent user preferences, with users as rows, items as columns, and ratings as the values within the matrix. The example below demonstrates a simple utility matrix with users and their ratings for various movies:

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.2	
Bob		0.5		0.3
Carol	0.2	1		
David			0.4	

Table 4: An example of a utility matrix in a recommender system. The goal is to predict the blank entries in the matrix.

The blanks in the matrix represent the unknown preferences of the users for certain items. The primary goal of a recommender system is to accurately predict these blanks, thereby personalizing item suggestions to the user's taste.

Key Problems in Recommender Systems

Recommender systems face several core challenges, which are essential to understand for their effective implementation:

1. **Gathering "Known" Ratings for Matrix:** The first challenge is to collect the data that populates the utility matrix. This data collection process is critical as it lays the foundation for how the system will make future predictions.

2. **Extrapolating Unknown Ratings from the Known Ones:** Once the known data is collected, the next step is to predict the unknown ratings. The system is particularly interested

in high unknown ratings, emphasizing discovering items that users are likely to favor. The focus is less on identifying dislikes and more on what the user might like.

3. **Evaluating Extrapolation Methods:** The final challenge is to evaluate the success or performance of the methods used to extrapolate the unknown ratings. This involves measuring how well the recommender system's predictions match up with the actual preferences of the users.

Addressing these problems effectively is crucial for the development of a recommender system that provides valuable and accurate recommendations to its users.

Gathering Ratings for Recommender Systems

Gathering ratings is the initial step in building the utility matrix for a recommender system. Ratings can be collected in two primary ways:

- **Explicit Rating Collection:**
  - Users are directly asked to rate items, providing clear and direct feedback on their preferences.
  - However, this method often encounters practical challenges as users may not be inclined to spend time rating items.
  - Crowdsourcing is one solution where individuals are paid to provide ratings, ensuring a consistent flow of data.
- **Implicit Rating Collection:**
  - Ratings are inferred from user actions, such as purchases or the time spent on content, under the assumption that these actions reflect positive ratings.
  - For example, if a user purchases an item, it can be assumed they would rate it highly.
  - The challenge arises in interpreting actions that might imply lower ratings.

Each method has its benefits and drawbacks, and the choice between them often depends on the specific requirements and context of the recommender system being developed.

Extrapolating Utilities

A primary issue in the construction of recommender systems is the sparsity of the utility matrix U. The matrix is often sparse because:

- Most users have not rated the majority of items, leading to many undefined values.
- New items often have no ratings, a situation known as the cold start problem.
- New users have no history of ratings, contributing to the cold start problem from another angle.

To address the sparsity issue and the associated cold start problems, recommender systems can employ several strategies:

1. **Content-based Approaches:** These methods use item features to recommend additional items similar to what the user has liked in the past.
2. **Collaborative Filtering:** This technique makes automatic predictions about the interests of a user by collecting preferences from many users.
3. **Latent Factor Based Methods:** These approaches, such as matrix factorization, identify latent factors from the observed interactions between users and items.

Each of these approaches offers a way to infer user preferences and predict missing values in the utility matrix, thereby overcoming the key challenges of sparsity and cold starts.

Content-based Recommendations

The essence of content-based recommendations is to suggest items that are similar to those a customer has previously expressed interest in. The main idea can be summarized as follows:

- Items are recommended to a customer x based on the similarity to previous items that x rated highly.

For example, in a movie recommendation scenario:

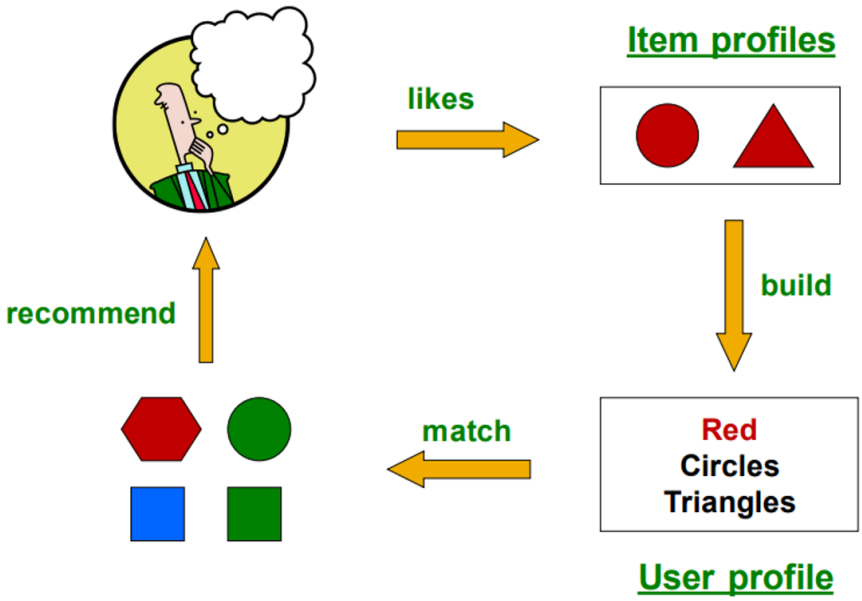
- Movies featuring the same actors, directors, or within the same genre as those liked by the customer are suggested.

The same principle applies to other domains such as:

- Websites, blogs, news articles: Recommending content with similar themes or subject matter to what the user has shown an interest in.

Content-based systems focus solely on the user's past behavior and the features of the items to generate recommendations, without taking into account the preferences of other users.

RecSys Operating Mode



Item Profiles

In content-based recommender systems, an item profile is crucial for determining the similarity between items and user preferences. Each item profile is defined as follows:

- It is a set (or vector) of features that represent the characteristics of the item.
  - For movies, these features could include the author, title, actors, directors, etc.
  - For text documents, the features might be a set of 'important' words within the document.
- The selection of important features can be informed by the TF-IDF (Term Frequency-Inverse Document Frequency) heuristic from text mining.
  - The term 'feature' refers to a measurable piece of data that is used to represent the aspect of the item being profiled.
  - The document equates to the item for which the profile is being created.

The process of creating item profiles is a fundamental step in content-based filtering as it directly impacts the recommendation quality by influencing how items are compared based on their attributes.

Sidenote: TF-IDF

(If calculation given, formula is given on test) Term Frequency-Inverse Document Frequency (TF-IDF) is a numerical statistic intended to reflect how important a term is to a document in a collection. The computations are as follows:

- f_{ij} is the frequency of term i in document j.
- The Term Frequency TF_{ij} is calculated as $\frac{f_{ij}}{\max_k f_{kj}}$, which normalizes the term frequency to account for the length of the document.
- n_i is the number of documents mentioning term i, and N is the total number of documents.
- The Inverse Document Frequency IDF_i is computed as $\log\left(\frac{N}{n_i}\right)$.

The TF-IDF score w_{ij} for a term in a document is then given by the product $TF_{ij} \times IDF_i$. The document profile, in the context of TF-IDF, is a set of words with the highest TF-IDF scores, which are considered the most relevant or significant terms in the document.

User Profiles and Prediction

In content-based recommender systems, constructing a user profile is key to making accurate predictions. User profiles can be formulated through various methods:

- A common approach is to compute a weighted average of the item profiles the user has rated.
- A variation involves weighting by the difference from the average rating for each item, which can help in accentuating preferences.
- ...

For prediction, a heuristic is employed to estimate the utility $u(x, i)$ of an item i for a user profile x:

$$u(x, i) = \cos(x, i) = \frac{x \cdot i}{\|x\|\|i\|}$$

Here, the cosine similarity measures the cosine of the angle between the user profile vector x and the item profile vector i, which correlates with the user's likely preference for the item.

Pros of Content-based Approach

Content-based recommendation systems come with several advantages:

- Independence from other users' data, eliminating cold-start or sparsity problems associated with new items or users.
- The capability to recommend items to users with unique tastes by focusing on individual preferences rather than crowd wisdom.

- The ability to recommend new and unpopular items since recommendations are based on content, not on item popularity.
- Provision of explanations for recommendations, as they can be traced back to specific content-features that an item has.

These aspects highlight the self-sufficiency and personalization strengths of content-based systems.

Cons of Content-based Approach

Despite its strengths, the content-based approach has some limitations:

- Identifying the appropriate features for accurate item representation can be challenging, particularly for complex items like images, movies, and music.
- Building user profiles for new users can be difficult as there is no prior data to analyze.
- Overspecialization can lead to a narrowed scope of recommendations, never suggesting items outside the user's established content profile.
- It fails to consider the quality judgments of other users, potentially missing out on popular items that a user might appreciate.

These limitations underscore the need for careful feature selection and the potential benefits of hybrid recommendation systems.

Collaborative Filtering

Collaborative filtering is a technique used by recommender systems to make automatic predictions about the interests of a user by collecting preferences from many users. The central premise of collaborative filtering is to exploit the quality judgments of other users to provide relevant recommendations. This method has several facets:

The Collaborative Filtering Process

The collaborative filtering process involves:

- Maintaining a database of users' ratings of a variety of items.
- Identifying users with similar preferences to the active user.
- Recommending items that were rated highly by these similar users but not yet rated by the active user.

This approach is widely utilized by many existing commercial recommenders such as Netflix, Amazon, and eBay.

Building a Recommendation Model

To build a recommendation model using collaborative filtering, the following steps are taken:

- Consider a user x.
- Find a set N of other users whose ratings are "similar" to x's ratings.
- Estimate x's ratings based on the ratings of users in N.

The similarity between users is typically measured using statistical techniques that compare their ratings patterns across the set of items.

Finding "Similar" Users

To identify users with similar interests in a collaborative filtering system, various similarity measures can be applied:

- **Jaccard Similarity Measure:** This method compares users based on the items they have both interacted with, regardless of the rating value. However, it may not be as effective for collaborative filtering since it ignores the ratings' values.
- **Cosine Similarity Measure:**
  - Defined as $\text{sim}(x, y) = \cos(\mathbf{r}_x, \mathbf{r}_y) = \frac{\mathbf{r}_x \cdot \mathbf{r}_y}{\|\mathbf{r}_x\| \|\mathbf{r}_y\|}$, where $\mathbf{r}_x$ and $\mathbf{r}_y$ are the rating vectors of users x and y, respectively.
  - Calculated using the formula $\text{sim}(x, y) = \frac{\sum_i r_{xi} \cdot r_{yi}}{\sqrt{\sum_i r_{xi}^2} \cdot \sqrt{\sum_i r_{yi}^2}}$, where r_{xi} and r_{yi} are the ratings of user x and user y for item i, respectively.
  - A potential problem with cosine similarity is that it treats missing ratings as negative values.
  - A potential solution to the cosine similarity problem is to normalize the ratings by subtracting the row mean, effectively centering the data around zero. This modification can make cosine similarity approximate the correlation when data is centered at 0.
  - This solution effectively makes the normalized cosine similarity and the Pearson Correlation Coefficient mathematically equivalent.
- **Pearson Correlation Coefficient:**
  - Measures the linear correlation between two users' ratings, accounting for the average rating of each user.
  - Calculated as $\text{sim}(x, y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)(r_{ys} - \bar{r}_y)}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)^2} \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \bar{r}_y)^2}}$, where S_{xy} is the set of items rated by both users, and $\bar{r}_x$, $\bar{r}_y$ are their average ratings.

The choice of similarity measure can significantly impact the performance of the recommendation system, and it often requires experimentation to find the most suitable one for a given application.

Rating Predictions

Once we have established a similarity metric between users, we can use it to predict ratings:

- Let r_x be the vector of user x's ratings.
- Let N be the set of k users most similar to x who have rated item i.

To predict the rating for an item s of user x, we can use the following formula:

$$r_{xi} = \frac{1}{k} \sum_{y \in N} r_{yi}$$

Alternatively, to incorporate the similarity weighting, we use:

$$r_{xi} = \frac{\sum_{y \in N} s_{xy} \cdot r_{yi}}{\sum_{y \in N} s_{xy}}$$

where s_{xy} is the similarity score between users x and y, and r_{yi} is the rating of item i by user y. This method predicts user x's rating for item i based on the weighted average of ratings from similar users.

There are many possible variations and improvements to this basic prediction approach, depending on the specific characteristics and goals of the recommender system.

Item-Item Collaborative Filtering

While we have so far considered user-user collaborative filtering, another perspective is the item-item collaborative approach:

- This model focuses on finding similar items rather than similar users. For a particular item i, the system identifies other items that are similar and uses their ratings to estimate the rating for item i.
- The estimation of the rating for item i is based on the ratings of the similar items. The same similarity metrics and prediction functions used in the user-user model can be applied here.
- The only difference is that we now look at items, so our matrix must be transposed or we must replace all row operations for column operations. We can see this from the corresponding ratings prediction formula for item-item CF given below.
- It has been observed in practice that item-item collaborative filtering often outperforms user-user collaborative filtering. This could be due to items typically having more consistent features than users have tastes, which can vary widely and be multifaceted.

Ratings prediction formula:

$$r_{ix} = \frac{\sum_{y \in N(i,x)} s_{iy} \cdot r_{yx}}{\sum s_{iy}} \tag{1}$$

The item-item approach provides an alternative to user-user filtering, addressing some of its limitations and improving recommendation quality in certain contexts.

Pros and Cons of Collaborative Filtering

Collaborative filtering, as a method for building recommendation systems, has its set of strengths and weaknesses:

Pros

- It works for any kind of item without the need for feature selection. This makes it a flexible approach that can be applied to various types of content.

Cons

- **Cold Start:** The system requires a sufficient number of users to function effectively. It faces difficulties when new users or items have insufficient interactions to establish similarities.
- **Sparsity:** The user-item interaction matrix is typically sparse, making it challenging to find users who have rated the same items.
- **First Rater Problem:** If no one has rated an item, the system cannot recommend it. This is particularly problematic for new or niche items.
- **Popularity Bias:** The system might favor popular items and may not recommend items that would appeal to someone with unique tastes. This could lead to a homogenization of recommendations.

These considerations highlight both the versatility and the limitations of collaborative filtering, indicating areas where improvement or supplementary techniques might be beneficial.

Hybrid methods

We could also try to use hybrid methods for recommender systems:

- implement two or more different recommenders and combine predictions
  - Perhaps using linear model
- Add content-based methods to collaborative filtering
  - Item profiles for new item problem
  - Demogrhapics to deal with new user problem

Cold Start Problem in Collaborative Filtering

Challenges

Collaborative Filtering faces a few initial challenges:

- **New Item Problem:** When an item is new and has been rated by a small number of users, making accurate predictions is challenging.

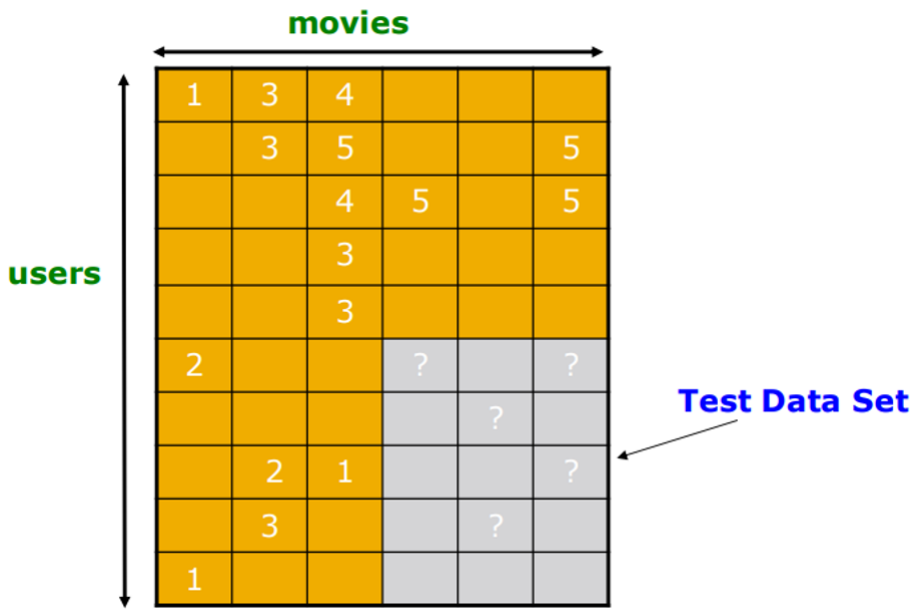
- **New User Problem:** A new user without a history of ratings may not have enough overlap with existing users to compute reliable similarities.
- **New Community Problem:** Without enough ratings, personalized collaborative filtering recommendations lack differentiation and cannot effectively leverage the system's full potential.

Possible Solutions to the Cold Start Problem

To overcome these challenges, the following solutions can be implemented:

- **For New Users:**
  - Provide non-personalized recommendations until enough data from the user is collected.
  - Encourage users to describe their preferences or use demographic information to match them with similar users' preferences.
- **For New Items:**
  - Employ non-CF techniques, such as content analysis or using metadata for recommendations.
  - Randomly select items with few or no ratings and encourage users to rate them to gather more data.
- **For New Communities:**
  - Establish a reward system to incentivize a subset of users to rate items, which can act as a bootstrap for the recommender system.

Evaluating Quality of Predictions and Recommendations



Assessing the performance of recommendation systems is crucial. The following metrics are commonly used:

- **Root-Mean-Square Error (RMSE):** Measures the average magnitude of the errors in the predictions, where r_{xi} is the predicted rating, and r^*_{xi} is the true rating of item i by user x.

$$RMSE = \sqrt{\sum_{x,i}(r_{xi} - r^*_{xi})^2}$$

- **Precision at Top 10:** Evaluates what percentage of the top 10 recommended items are actually relevant to the user.
- **Rank Correlation:** Utilizes Spearman's correlation to compare the ranking order of the items provided by the recommendation system against the actual order preferred by the user.
- **Coverage:** Refers to the number of items or users for which the system can provide predictions.
- **Precision:** Concerns the accuracy of the predictions, assessing whether the recommended items meet the users' preferences.

Each metric provides different insights into the recommendation system's performance and can be used to guide the improvement of the system.

Mining Social Networks

Mining social networks involves the extraction of actionable patterns and insights from data generated in social media platforms. This field intersects numerous disciplines including computer science, sociology, statistics, and data science. It focuses on analyzing social structures through the use of networks and graph theory. Central tasks include identifying influential users, understanding social interactions, community detection, sentiment analysis, and spreading of information. The insights gained can power applications like targeted marketing, political campaign strategies, and social recommendations. As social media generates vast amounts of data daily, the challenges of scalability and real-time processing are significant. Advanced techniques from big data, such as machine learning algorithms and data mining tools, are employed to handle, analyze, and interpret this data effectively.

Analysis of Large Graphs: Community Detection

Community detection in large graphs is a fundamental aspect of mining social networks. This process involves identifying natural groupings or communities within large networks, which can be crucial for understanding the relationships and interactions between different users or entities. By analyzing these communities, we can gain insights into social dynamics, group behaviors, and collective interests.

Understanding Social Network Graphs

A social network graph is a visual representation of the relationships and interactions among individuals within a network. Nodes represent individuals or entities, and edges represent the connections between them. These graphs can become immensely complex with the increasing size of the network, necessitating sophisticated analytical techniques to discern patterns and structures.

Figure 43: A complex social network graph with various interconnections.

Networks and Communities

Communities in social networks often manifest as groups of nodes with denser connections internally than with the rest of the network. The organization of networks into modules, clusters, or communities can significantly inform the understanding of network functions and the roles of individual nodes.

Goals of Community Detection

The primary goal of community detection is to find densely linked clusters within the network. These clusters can correspond to real-world structures like friendship circles, professional groups, or interest-based communities, providing valuable insights for targeted marketing, recommendation systems, and social research.

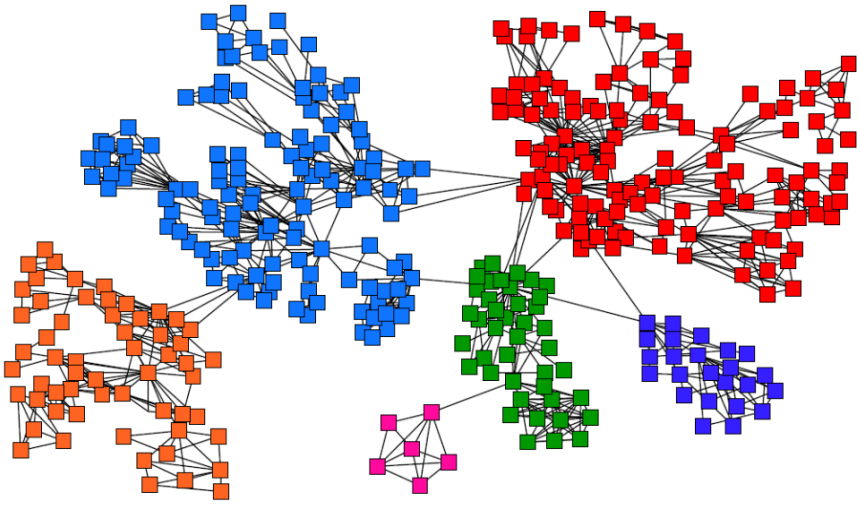


Figure 44: Graphical representation of the goal to find densely linked clusters in a network.

Social Circles and Trust in Networks

Discovering social circles and understanding the circles of trust within networks like Twitter and Facebook can reveal the strength and depth of social bonds. Trust levels vary among different social circles, and analyzing these can help in designing systems that account for user privacy and information flow within the network.

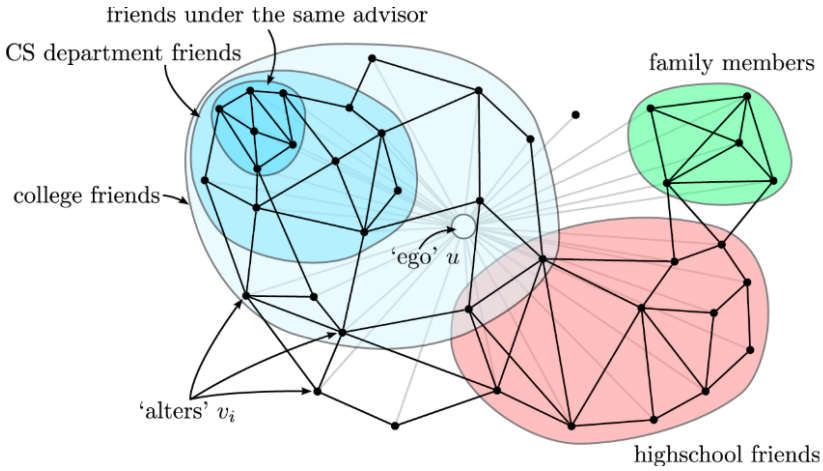


Figure 45: Identifying social circles and trust levels in a Twitter and Facebook graph.

Methods for Finding Communities

To uncover communities in social networks, we must apply algorithms designed to detect densely connected subgraphs within a larger network. Communities are typically characterized by a high density of edges among the nodes within the same community compared to those outside it. For this purpose, we will focus on undirected (unweighted) networks where connections represent a mutual relationship, without consideration of their strength or direction.

The algorithms for community detection range from hierarchical clustering to optimization of a modularity function. These techniques strive to maximize the number of within-community edges while minimizing the between-community edges. Understanding the methodology for community detection is crucial for interpreting the results and applying them in practical scenarios such as social media analysis, epidemiology, and marketing.

Hierarchical Clustering in Community Detection

Hierarchical clustering is a versatile method for detecting nested clusters within data, applicable to various domains, including social network analysis. This approach builds a hierarchy of clusters by either recursively merging smaller clusters into larger ones (agglomerative approach) or by splitting larger clusters into smaller ones (divisive approach). This process can be visualized with a dendrogram, a tree-like diagram that records the sequence of merges or splits and illustrates the multi-level structure of clustering.

However, when applying traditional hierarchical clustering techniques to graph data, such as social networks, several challenges arise. The absence of a well-defined distance measure between clusters can lead to suboptimal clustering results. Furthermore, there is a high risk of improperly merging two distinct communities, leading to a loss of meaningful structure within the data. These limitations must be addressed through the development of graph-specific clustering techniques or the adoption of alternative methods that are better suited to the structure of network data.

The Strength of Weak Ties and Edge Betweenness

The concept of the strength of weak ties is critical in the analysis of social networks. It suggests that weaker ties, which often form bridges between different communities, can be more influential than stronger ties in the spread of information through a network. One way to quantify the significance of a tie is through edge betweenness.

Edge betweenness is defined as the number of shortest paths that pass through an edge. Edges with high betweenness are typically the ones that connect communities, acting as bridges, and their removal can often result in the formation of distinct groups. This intuition is particularly valuable when trying to understand communication patterns, such as call volumes in telecommunication networks, and identifying key connectors or bottlenecks within a network.

The visualization of edge strengths, such as call volumes, can reveal the intensity of connections in a real network. Similarly, mapping edge betweenness can highlight the critical pathways through which information flows. Understanding these metrics allows for the strategic planning of information dissemination and can inform the design of robust network infrastructures.

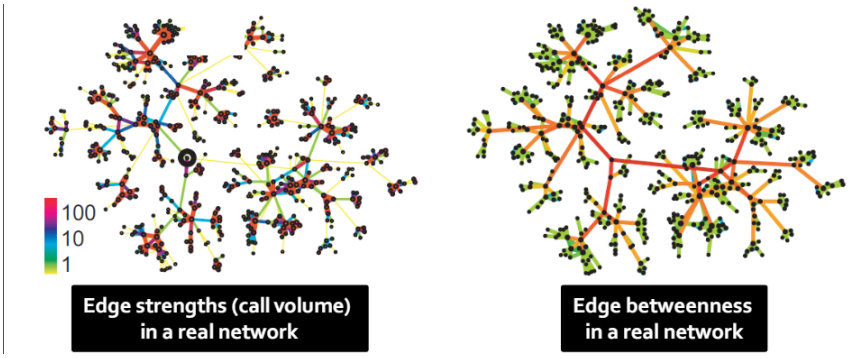


Figure 46: Comparison of edge strengths and edge betweenness in a network, highlighting the importance of weak ties.

Girvan-Newman Algorithm for Community Detection

The Girvan-Newman algorithm represents a divisive hierarchical clustering method that leverages the concept of edge betweenness. This algorithm operates under the premise that the most significant edges in a network are those that form bridges between communities. The Girvan-Newman method systematically removes these edges to uncover the underlying community structure.

To apply the Girvan-Newman algorithm:

1. Calculate the betweenness for all edges in the network.
2. Remove the edge with the highest betweenness.
3. Recalculate betweenness for all edges affected by the removal.
4. Repeat the process until no edges remain.

As the algorithm progresses, the removal of edges with high betweenness scores eventually disconnects the network into distinct communities. This process provides a hierarchical decomposition of the network, revealing both the macro- and micro-scale community structures. The Girvan-Newman algorithm is particularly well-suited for undirected and unweighted networks and has been pivotal in the field of network science.

Girvan-Newman Algorithm: A Practical Example

This example illustrates the necessity of recalculating edge betweenness after each removal, as the shortest paths—and thus the betweenness scores—can change. The result is a hierarchical decomposition of the network into communities, showing the clusters at various levels of granularity.

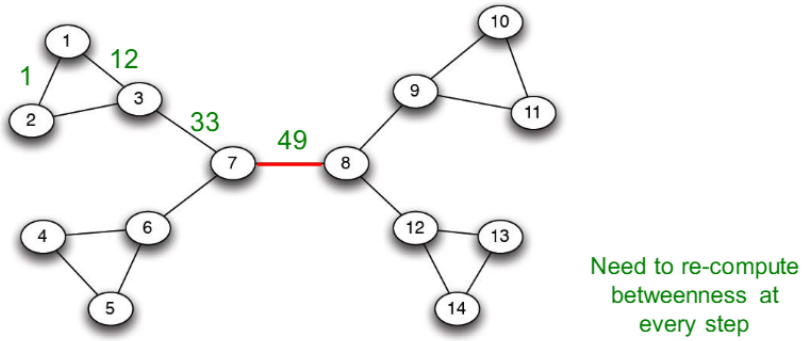


Figure 47: Initial step of the Girvan-Newman algorithm showing edge betweenness calculation.

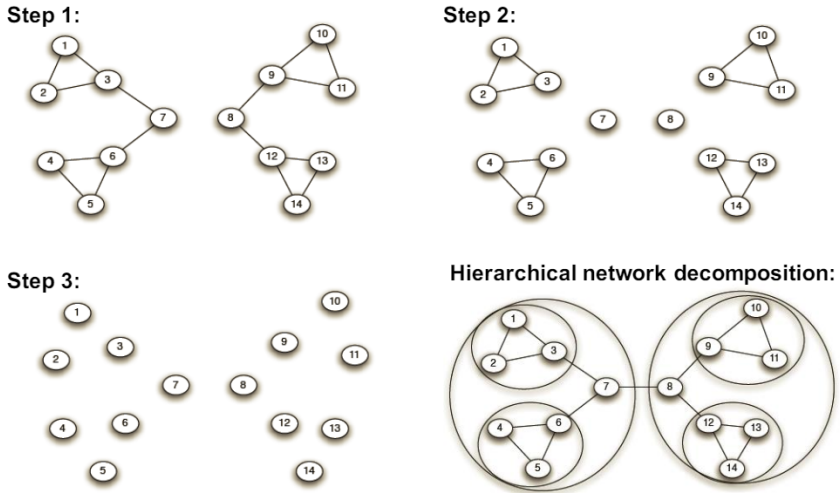


Figure 48: Subsequent steps of the Girvan-Newman algorithm and the resulting hierarchical network decomposition.

Computing Edge Betweenness in Network

To compute the betweenes of all the edges in the network, we employ the breadth-first-search (BFS). The procedure goes as follows:

- Perform a BFS starting from a chosen node to find the shortest paths to all other nodes.
- Then, starting from the leaves of the BFS tree and moving upwards, calculate the node flow. The flow of a node is one plus the sum of the flows of its child nodes.

- As the flow is propagated upwards, it is divided among the edges leading to the parent nodes. If an node has multiple shortest paths, the flow is split proportionally based on the number of shortest paths going to the node. Thus, the edges involved only get a proportion of the nodes flow.
- Complete this process for all nodes in the network, starting from each node as the root of the BFS tree. We determine the edge betweenes by summing the flow values across all the BFS trees for each edge and dividing the results by two to account for each shortest path being counted twice.

This method ensures that the calculated betweenes captures the frequency and distribution of shortest paths across the network.

Girvan-Newman Algorithm: An Example

To illustrate the Girvan-Newman algorithm in action, consider the following example:

1. Start with a connected graph and calculate the betweenness centrality for all edges.
2. Identify the edge with the highest betweenness and remove it from the graph.
3. Recalculate betweenness centrality for all remaining edges affected by the removal.
4. Repeat steps 2 and 3 until no edges remain or the desired number of communities is achieved.

The process of recalculating betweenness centrality is crucial after each edge removal because the shortest paths—and thus the betweenness centrality of the remaining edges—may change as the network structure changes.

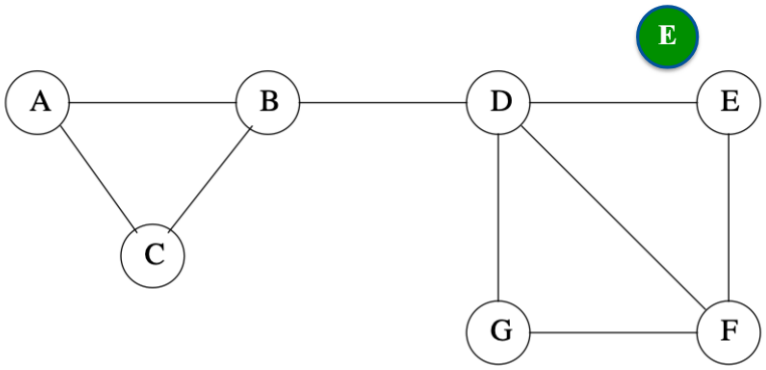


Figure 49: Initial network structure for the Girvan-Newman algorithm example.

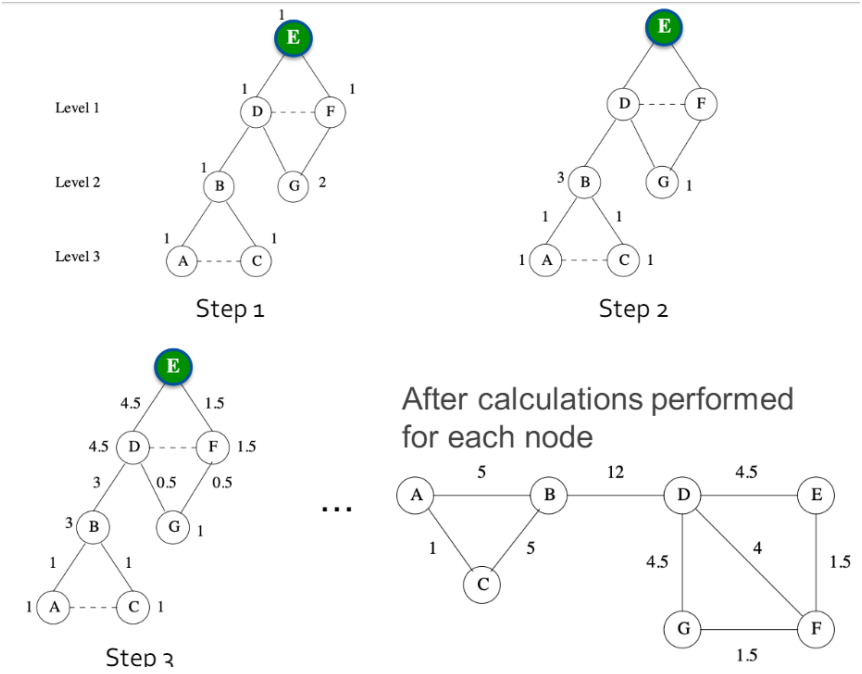
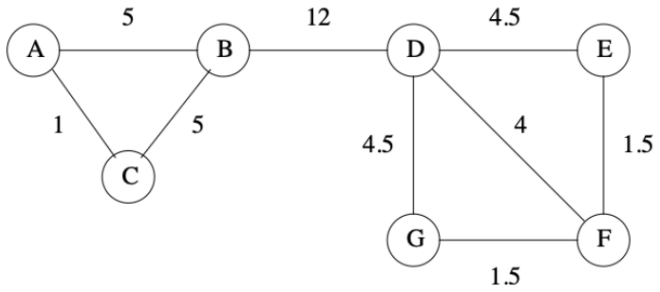


Figure 50: The Girvan-Newman algorithm in action: Edge betweenness is recalculated after each edge removal.



Repeat until no edges are left:

(Re)Calculate betweenness of edges (only needed for affected edges)
Remove edges with highest betweenness

I.e.: First B-D, then A-B or B-C, etc.

Figure 51: Final network structure after successive edge removals in the Girvan-Newman algorithm.

This algorithm effectively reveals the community structure within a network by highlighting the edges that play significant roles in connecting different groups. As edges are removed, the most "central" connections are identified, leading to an understanding of the network's modular structure.

Understanding Network Communities and Modularity

Communities within networks are characterized by sets of nodes that are more densely connected to each other than to the rest of the network. To quantify the strength of a community structure, we use modularity Q, which is:

- A measure of the strength of division of a network into communities. High modularity indicates dense connections within communities and sparse connections between them.
- Calculated for a given partitioning of the network into groups S, with the modularity Q being proportional to the difference between the actual number of edges within groups and the expected number of such edges under a null model.

A null model is a reference model that assumes no community structure. It provides a benchmark against which the actual edge density within communities is compared. In the context of modularity, the null model predicts the number of edges we would expect to see between nodes within a community purely by chance, given the network's node degrees. By comparing the observed edge density to that predicted by the null model, we can infer whether the community structure is statistically significant or could merely be a result of random chance.

Defining Network Communities through Modularity

The network's modularity Q is a measure that reveals the strength of the division of a network into communities. We define modularity as follows: Given a partitioning of the network into groups $s \in S$:

$$Q \propto \sum_{s \in S} [\#\text{ edges within group } s - (\text{expected } \#\text{ edges within group } s)]$$

Null Model for Networks: The Configuration Model

When analyzing network data to find patterns that are not just random noise, a null model is a crucial tool. It's a randomized version of the original network that preserves some of its properties but otherwise has no structure. In network community detection, the null model helps us to define and measure modularity by comparing the actual density of within-community edges to what would be expected if connections were made at random.

The Configuration Model is a specific type of null model that retains the degree distribution of the original network but randomizes the connections. Given a real network G with n nodes and m edges, the Configuration Model creates a rewired network G' by:

- Keeping the same degree for each node as in the original network.
- Connecting nodes randomly, thus destroying any existing community structure without altering node degrees.

- Allowing for the possibility of multiple edges between two nodes (multigraph) which may occur due to the random connection process.

The expected number of edges between nodes i and j in this model, assuming nodes i and j have degrees k_i and k_j respectively, is given by:

$$\frac{k_i k_j}{2m}$$

This formula calculates the probability of an edge existing between two nodes based on their degrees and is used in the modularity function to determine the strength of the network's division into communities.

Quantifying Community Structure with Modularity

Modularity Q is a scalar value between -1 and 1 that measures the density of links inside communities as compared to links between communities. In the context of a given partitioning S of a graph G, modularity is defined as:

$$Q(G, S) = \frac{1}{2m} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} \left(A_{ij} - \frac{k_i k_j}{2m} \right)$$

where:

- A_{ij} is an element of the adjacency matrix, which is 1 if there is an edge between nodes i and j, and 0 otherwise.
- k_i and k_j are the degrees of nodes i and j, respectively.
- m is the total number of edges in the graph.

This formula essentially compares the actual density of edges in a community to the density that would be expected if the graph were random and had the same degree distribution. A positive modularity indicates that the number of edges within groups exceeds the expected number, suggesting a strong community structure. Specifically, modularity values between 0.3 and 0.7 typically indicate significant community structure within the network.

The modularity cost, normalized to fall within the range of -1 to 1, penalizes the fragmentation of the network and rewards the formation of dense communities, thereby guiding the optimization process used in community detection algorithms.

Using Modularity to Determine the Number of Clusters

Modularity not only measures the strength of a community structure within a network but also serves as a guiding metric for determining the optimal number of clusters or communities. By maximizing the modularity score, we can identify the partitioning of the network that best delineates the community structures. Here's why modularity is particularly useful for this purpose:

- **Objective Measure**: Modularity provides an objective criterion to compare different partitionings. The partitioning that yields the highest modularity is considered the best division of the network into communities.

- **Resolution Limit**: Modularity takes into account the resolution limit, which can sometimes lead to a preference for a certain number of communities. It allows for the comparison of the quality of the divisions rather than just the quantity.
- **Quantitative Analysis**: Instead of relying on a subjective assessment of the community structure, modularity offers a quantitative method to assess the division of the network into communities.

A common approach to using modularity in determining the number of clusters involves performing hierarchical clustering and then examining the modularity at each level of the hierarchy. The dendrogram of hierarchical clustering can be cut at different heights to yield different numbers of clusters. The modularity score associated with each cut can then guide the selection of the number of clusters: a higher modularity indicates a more accurate reflection of the inherent community structure.

The plot of modularity values against different numbers of communities often exhibits a peak at the optimal number of clusters. This is the point where the communities are neither too granular nor too broadly defined, striking a balance between the detail and the cohesiveness of the community structure.

Speeding Up Betweenness Centrality Computation

For a graph consisting of n nodes and e edges, the time complexity to compute the betweenness centrality for all edges is $O(ne)$, which can become impractical for large-scale networks. To mitigate the computational demands, an approximate solution can be employed:

- **Random Sampling**: Instead of calculating betweenness centrality for every node, a subset of nodes is selected randomly.
- **Breadth-First Searches (BFS)**: Use the nodes from the random subset as starting points for BFS to estimate the betweenness centrality.

This approximation offers a trade-off between accuracy and computational efficiency. By using a representative sample, it is often possible to obtain a sufficiently accurate estimation of betweenness centrality, which can be adequate for most applications. This method dramatically reduces the time required for computation while still providing valuable insights into the network's structure.

Streaming Systems and Storm

Introduction to Streaming Systems

In the era of big data, the capability to process and analyze data in real-time as it flows into systems is invaluable. Streaming systems are designed to handle continuous, high-velocity streams of data, providing insights almost instantaneously. This real-time processing capability is critical in a variety of applications, from financial market analysis to social media, where the ability to react quickly to new information can provide a competitive edge or enhanced user experience.

The Role of Storm in Data Stream Processing

Among the various platforms available for streaming data, Storm has been recognized for its robustness and scalability. Storm enables the processing of data in real time, ensuring that each message is processed reliably. It operates by distributing data streams across a cluster of machines, processing each unit of data in a fault-tolerant manner. The goal of using Storm and similar systems like Spark Streaming and Flink is to comprehend and manage the ongoing deluge of information generated by sources such as social media, sensors, and online transactions.

Understanding Data Streams Through Visualization

An illustrative way to comprehend the scale of data generated every minute is to look at the activities across various platforms on the internet. For instance, in 2021, every Internet minute witnessed millions of messages sent, hundreds of thousands of hours of content uploaded to YouTube, and significant online spending. This immense volume of data creation calls for efficient streaming systems that can handle such scale and velocity.

Big Data Streaming Challenges and Solutions

The challenges posed by such vast amounts of data are not only in the storage but also in the speed of processing. Streaming systems like Storm are designed to tackle these challenges by providing a distributed computing platform specifically optimized for high-speed processing of streaming data. They enable businesses and organizations to react to incoming data in real time, making decisions and adjustments much more rapidly than traditional batch processing systems could allow.

As we delve deeper into the functionality and architecture of Storm, we will explore how it can be employed to extract meaningful information from the ceaseless stream of big data, ensuring that organizations remain agile and responsive in the fast-paced digital landscape.

Figure 52: The magnitude of data generated on the internet every minute.

Rationale for Data Stream Systems

Data stream systems are engineered to meet the demands of real-time data processing. The ability to obtain immediate insights from data as it is generated has become critical across various domains. Let's explore the compelling reasons for the adoption of these systems:

Real-Time Data Processing

Real-time views of data are necessary for:

- Tracking trends on social networks like Twitter.
- Gathering instant website statistics, as with Google Analytics.

- Operating intrusion detection systems in data centers.
- Financial monitoring and conducting analysis, such as for credit card fraud detection.

High Throughput and Low Latency

The capacity to process large volumes of data swiftly, sometimes in a matter of seconds, and with high throughput, makes data stream systems indispensable. They can handle the continuous inflow of data, ensuring timely analysis and response.

Limitations of Batch Processing

Traditional batch processing systems like MapReduce are not designed for real-time analytics. They require the completion of the entire dataset's processing before deriving insights, which is infeasible for long-running or perpetual stream processing.

Significance of Streaming Systems

In contrast to batch processing, streaming systems do not wait for data collection to complete before starting the analysis. This enables organizations to detect patterns, anomalies, or trends as they happen, leading to more agile decision-making processes and immediate action.

With these considerations, streaming systems like Storm are not just a preference but a necessity for modern data architectures that require continuous and instantaneous data handling.

Handling Streaming Data

Streaming data poses unique challenges and opportunities for real-time data processing systems. Let's outline the key aspects of dealing with streaming data:

Continuous Processing

Streaming data requires continuous processing, which means that as data is generated, it is immediately ingested, aggregated, and analyzed without waiting for batch collection. This allows for instantaneous insights and actions based on the latest information.

Processing Model

The processing of streaming data can be represented as a Directed Acyclic Graph (DAG), where each node represents a processing step (like filtering, aggregating, or joining), and edges represent the flow of data. This model provides a clear and structured way of defining the data flow and the transformations that need to be applied.

Message Processing

Each message or piece of data is processed individually, allowing for a granular level of analysis. This one-at-a-time processing model is essential for tasks where the order and individual characteristics of each data point are critical.

Reliability and Fault Tolerance

To ensure reliability, it is necessary to implement mechanisms to handle failures. In streaming systems, this often means the ability to replay, restore, or restart processing streams from a known good state, ensuring that no data is lost and that processing can continue seamlessly after a system failure.

The capability to effectively manage streaming data is vital for systems that require up-to-the-minute analysis, such as financial tickers, social media feeds, and sensor networks in the Internet of Things (IoT). As we delve into specifics with systems like Storm, we will further understand how these principles are implemented in practice.

Delivery Guarantees in Streaming Systems

The reliability of data delivery is a cornerstone of streaming systems. There are three main types of delivery guarantees that such systems aim to provide:

At most once — Each message is delivered once or not at all. This guarantee ensures that there will be no duplicate processing of messages, but some messages may be lost. It is typically used in systems where missing some data does not have a critical impact.

At least once — Messages will definitely be delivered, but there is a possibility of the same message being delivered more than once. This approach is often acceptable when it is crucial that no messages are lost, and the system is designed to handle or ignore duplicates.

Exactly once — This is the most desirable but the most challenging to achieve, ensuring that each message is delivered and processed exactly once. It eliminates the problems of lost data and duplicate messages. However, implementing exactly-once semantics involves trade-offs, typically in the form of increased complexity and reduced performance due to the additional coordination required to prevent duplication or loss.

The choice of delivery guarantee impacts system design and performance. While exactly-once delivery is ideal, achieving it in a distributed system is non-trivial due to issues like network failures and the complexities of distributed transactions. It often requires intricate mechanisms such as two-phase commits, distributed consensus, or idempotent operations.

In practice, the choice between these guarantees is governed by the specific requirements of the application and the consequences of data loss or duplication.

Models of Stream Processing

Stream processing can be implemented using various models, each with its unique approach to handling continuous data flows.

Traditional Stream Processing Systems

Traditional stream processing systems operate on a continuous operator model, where each record is processed sequentially, one at a time. This model is characterized by a pipeline of operators, where each operator performs a specific function such as filtering, transformation, or aggregation.

- **Source Operator**: Ingests data records into the system.

- **Continuous Operator**: Processes each record individually as it passes through.
- **Sink Operator**: Outputs the processed records to a storage system or another application.

Native Streaming (e.g., Storm, Flink)

Native streaming systems like Storm and Flink are designed to handle streams natively, processing records individually but with the capability to process many records in parallel across a distributed system. They offer low latency and can provide strong guarantees like exactly-once processing semantics.

Micro-Batching (e.g., Spark Streaming)

Spark Streaming employs a micro-batching approach known as discretized stream processing. In this model:

- Incoming records are grouped into small batches.
- Each batch is processed as a Resilient Distributed Dataset (RDD), which is Spark's abstraction for an immutable collection of objects that can be processed in parallel.
- Tasks within each batch are distributed across the cluster for processing.

While micro-batching introduces some latency compared to native streaming, it simplifies the processing model by leveraging the batch processing capabilities of Spark, enabling it to seamlessly integrate with batch processing and interactive queries.

These different models reflect the evolution of stream processing systems from handling records one at a time to processing large volumes of data in parallel, offering various trade-offs between latency, throughput, and fault tolerance.

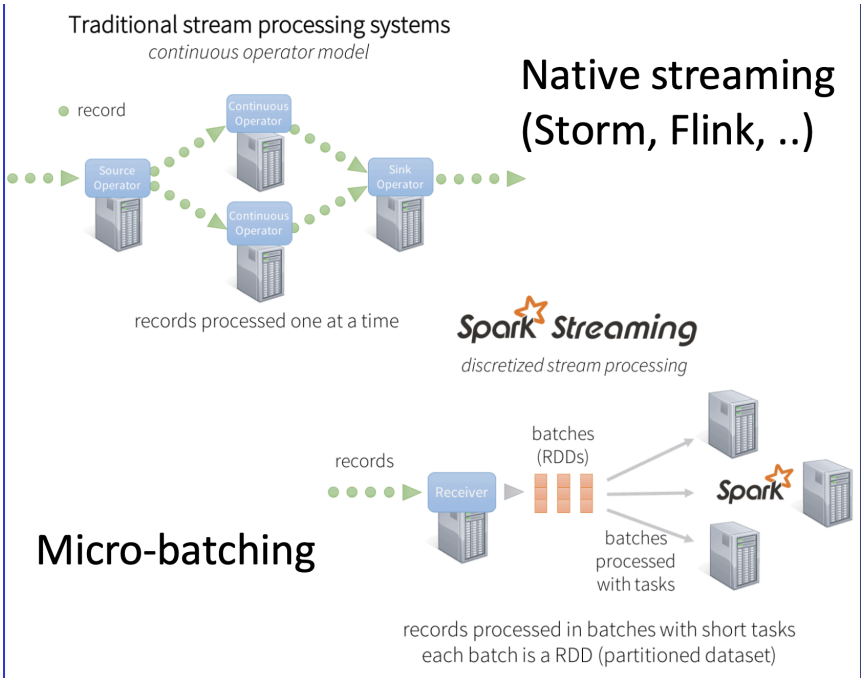


Figure 53: Comparison of traditional stream processing, native streaming, and micro-batching models.

Apache Storm: A Foundation for Real-time Analytics

Apache Storm has emerged as a prominent framework for real-time analytics, offering a scalable, fault-tolerant infrastructure designed to handle massive streams of data with low latency. Below are some key insights about Apache Storm:

- **Origins**: Storm was initially created by Nathan Marz at BackType and was later acquired by Twitter in 2011, underlining its effectiveness in processing real-time social media data.
- **Programming Language**: It is written in Clojure, a modern, functional, and expressive dialect of Lisp on the Java platform.
- **API**: Storm offers a simple yet powerful API that facilitates the development of complex real-time data processing tasks.
- **Language Support**: Storm's API supports multiple programming languages, making it accessible to a wide range of developers. Languages supported include Java, Python, and Ruby, among others.
- **Delivery Guarantees**: It provides robust delivery guarantees, supporting at-most-once, at-least-once, and even exactly-once processing semantics with Trident, an extension of Storm.

- **Real-world Applications**: Numerous companies leverage Storm's capabilities, with Twitter using it for personalization and search, and Spotify for music recommendation, monitoring, analytics, and ad targeting.

Apache Storm's real-time processing capabilities make it a versatile choice for various applications, from tracking user behavior on social networks to performing complex data analytics in real-time. The simple API and support for multiple languages facilitate a broad adoption, making Storm an integral part of modern data architectures.

Storm Components

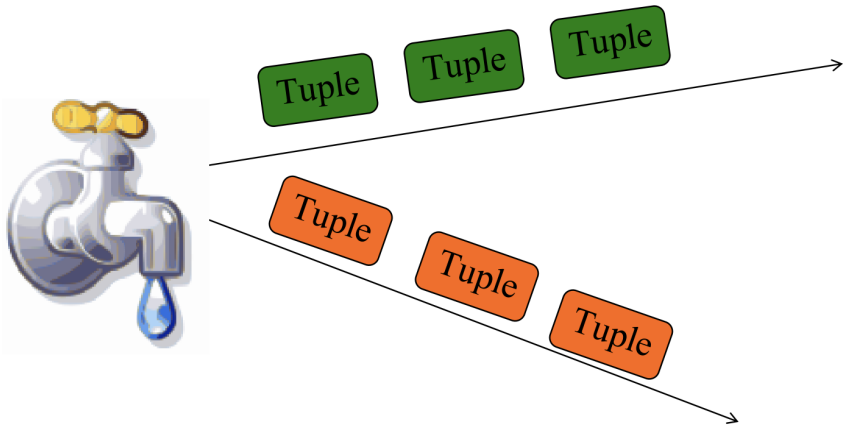
Stream processing is a fundamental aspect of real-time big data handling. Apache Storm stands out as one of the preferred systems for this purpose, providing various components that collectively contribute to efficient stream processing. In this section, we discuss the primary components that constitute Apache Storm, accompanied by illustrative figures to enrich the comprehension.

Tuples and Streams

- **Tuples:** A tuple in Apache Storm represents an ordered list of elements, serving as the fundamental data structure that Storm's components produce and process. For instance, a tuple could be denoted as <tweeter, tweet>, which symbolizes a user and their corresponding message on a social media service.
- **Streams:** A stream is characterized by a sequence of tuples, potentially infinite in length. Storm processes these streams to distill actionable insights or to execute real-time data analytics.

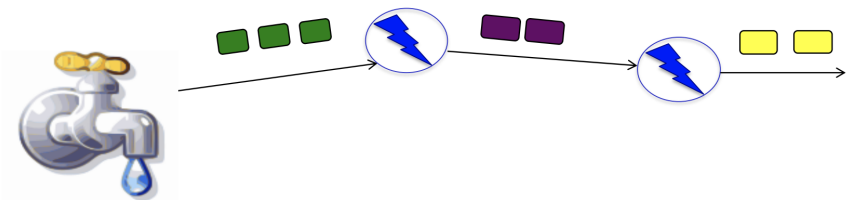
Spouts

Functioning as the origin of streams within a Storm topology, spouts interact with the data source, such as interfacing with the Twitter streaming API, and dispatch tuples into the stream for subsequent processing.



Bolts

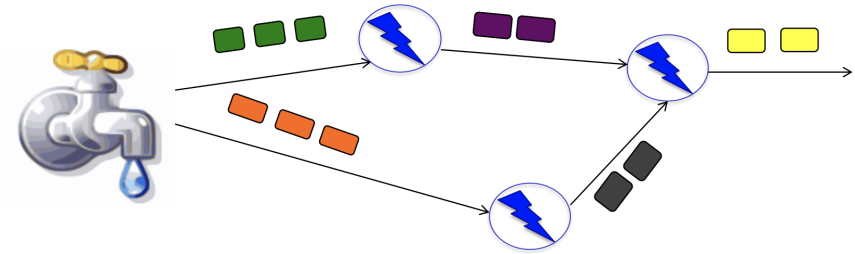
Bolts are the processors within Storm. They accept input streams sourced from spouts or other bolts, apply designated processing logic, and can emit new streams. Bolts' capabilities extend from executing functions, filtering tuples, to data aggregation and beyond.



Topologies

The topology in Storm is depicted as a directed graph comprising spouts and bolts, each node representing a processing element, while the edges signify the passage of tuples. This structure delineates the data transformation, accumulation, and storage procedures.

This delineation and interlinking of spouts and bolts dictate the data workflow and the resultant application conduct. Given the intrinsic parallelism of topologies, Storm is an efficacious tool for scalable real-time computations.



ZooKeeper Overview

Apache ZooKeeper is an open-source server which enables highly reliable distributed coordination. It is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and group services over large clusters in distributed systems.

- **Coordination Service:** ZooKeeper provides a coordination service for distributed applications. It exposes a set of simple primitives that distributed applications can build upon to implement higher-level services for synchronization, configuration maintenance, and groups and naming.
- **Availability and Scalability:** It is designed to be highly available and scalable to meet the demands of large-scale distributed systems.
- **Client-Server Model:** Each ZooKeeper server serves clients in the following manner:
  - Clients connect to exactly one server to submit their requests.

– Read requests are serviced from the local replica of each server's database, ensuring quick response times.
– Write requests, on the other hand, are processed by an agreement protocol to ensure consistency across server replicas.

This service is crucial in a variety of distributed applications as it guarantees simple and reliable primitives that are necessary for synchronization, configuration maintenance, and more.

Storm's System Architecture

Storm's system architecture consists of several components as shown in Figure 54.

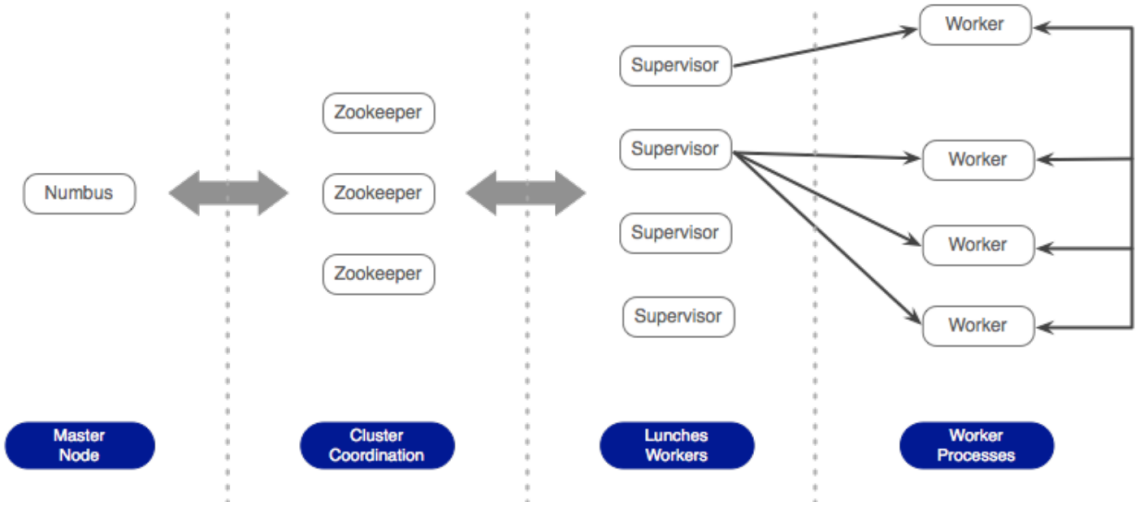


Figure 54: Storm's System Architecture

- **Nimbus:** Nimbus is the master node of the Storm cluster, which is responsible for distributing data among all the worker nodes, assigning tasks to worker nodes, and monitoring failures.
- **ZooKeeper:** ZooKeeper is a service used by a cluster (group of nodes) to coordinate between themselves and maintain shared data with robust synchronization techniques. Since Nimbus is stateless, it depends on ZooKeeper to monitor the working node status. In addition, ZooKeeper helps the supervisors to interact with Nimbus.
- **Supervisors:** Supervisors are the nodes that follow instructions given by the Nimbus. A supervisor has multiple worker processes and it governs worker processes to complete the tasks assigned by the Nimbus.
- **Worker:** A worker will execute tasks related to a specific topology. A worker process will not run a task by itself; instead, it creates executors and asks them to perform a particular task.

Understanding a Running Topology: Worker Processes, Executors, and Tasks

The operational backbone of a Storm topology is its runtime structure which consists of worker processes, executors, and tasks.

- **Worker Processes:** A machine in the Storm cluster may host one or more worker processes. Each worker process is associated with a particular topology and is responsible for executing a segment of it.
- **Executors:** Within each worker process, there can be multiple executors. An executor is essentially a thread spawned by the worker process. Every executor is dedicated to a specific component of the topology (either a spout or bolt).
- **Tasks:** Tasks are the atomic units of processing in Storm, and each task carries out the actual data processing work. An executor may run one or multiple tasks, but all tasks under a single executor will belong to the same topology component.

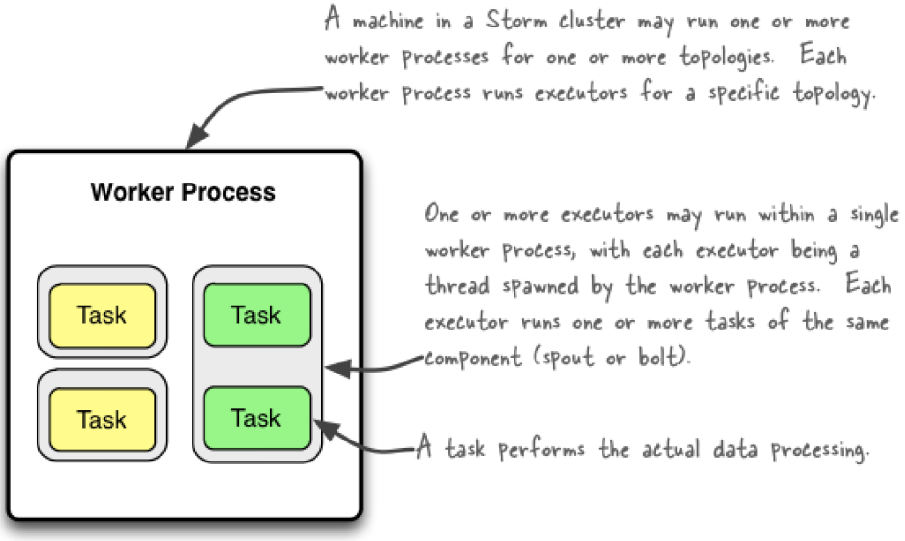


Figure 55: The structure of a worker process in a Storm cluster, depicting the relationship between worker processes, executors, and tasks.

These components work in concert to ensure that the Storm topology is able to process the stream of data efficiently and reliably. The parallelism afforded by the worker processes and executors enables Storm to handle large volumes of data with ease.

Storm Grouping Strategies

Grouping in Apache Storm defines how tuples are distributed to the tasks of a bolt. There are several grouping strategies, each serving a unique purpose based on the requirements of the topology.

- **Shuffle Grouping:** This strategy randomly partitions the tuples across the bolt's tasks, which helps in load balancing.
- **Fields Grouping:** Tuples are partitioned based on the hash values of specified tuple fields, ensuring that all tuples with the same field values are sent to the same task.
- **All Grouping:** The all grouping replicates the entire stream to all tasks of the bolt, which can be useful for state replication.
- **Global Grouping:** Global grouping sends all tuples of the stream to a single task, typically used for aggregating results.

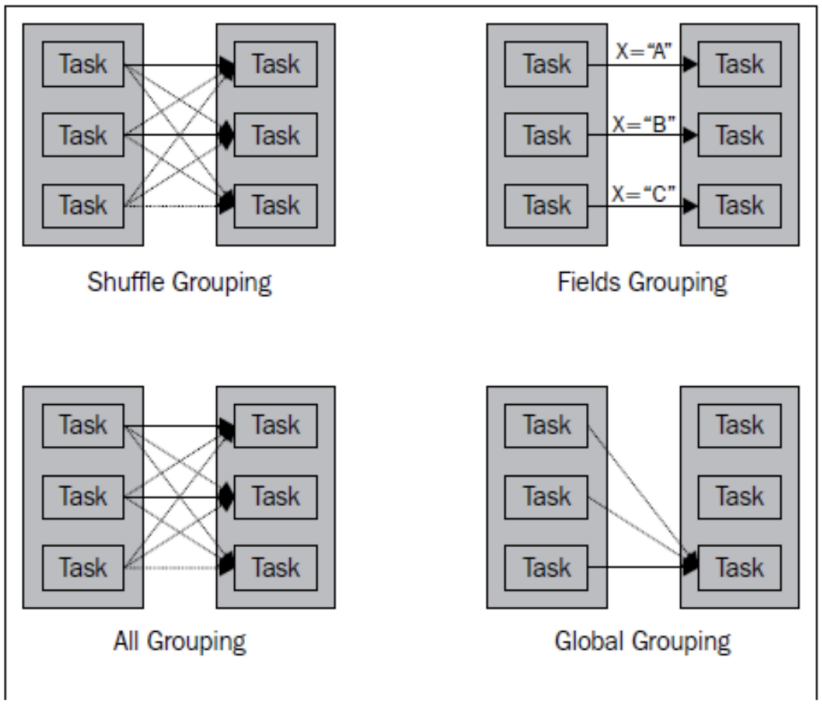


Figure 56: Illustration of different Storm grouping strategies, showing how tuples are distributed to tasks.

These grouping strategies are critical in determining how data flows through a Storm topology and can significantly impact the system's performance and scalability.

At-Least-Once Delivery in Storm

Achieving at-least-once delivery semantics ensures that no tuples are lost in the event of failures, which is critical for fault-tolerance in stream processing applications. The figure below illustrates a simplified Storm topology configured to provide such guarantees.

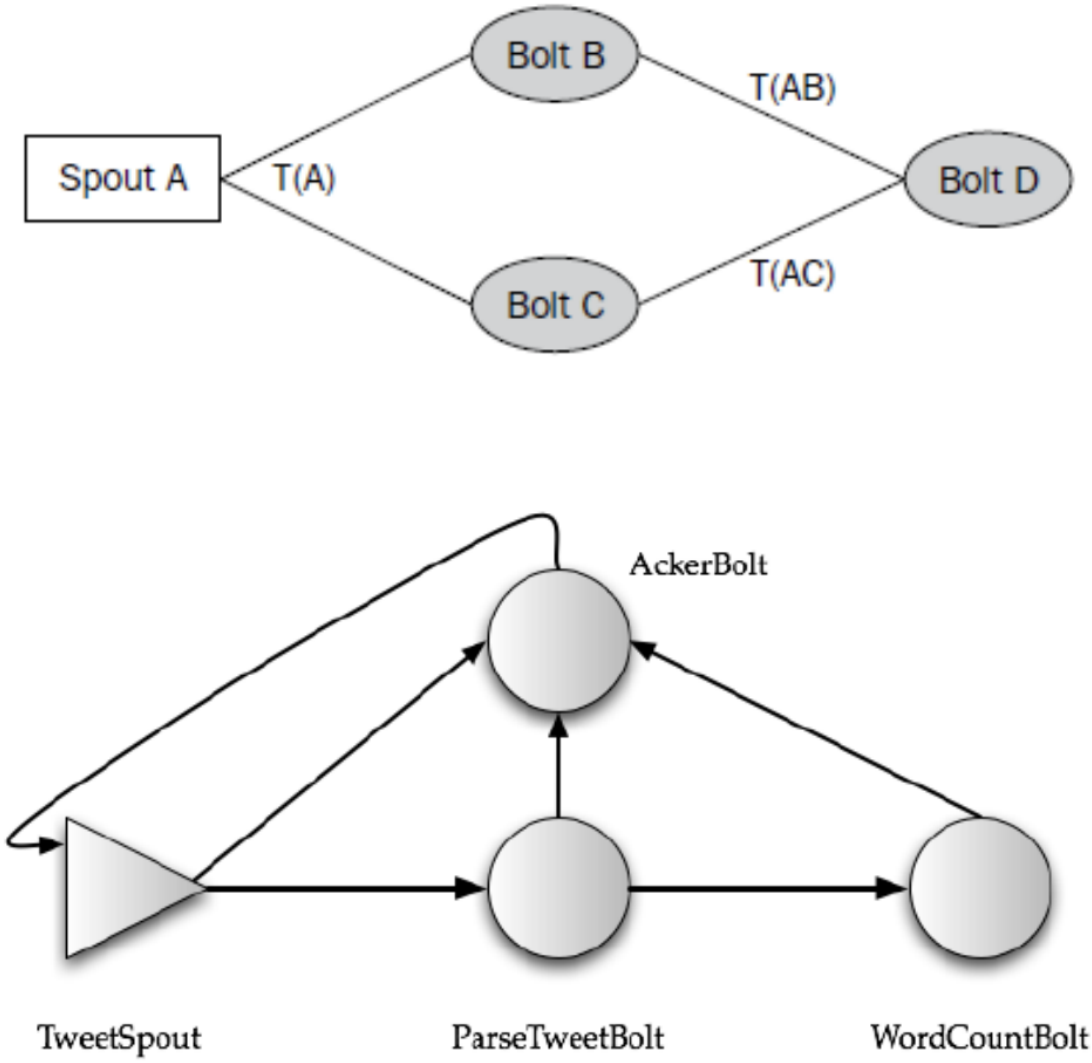


Figure 57: A simplified representation of achieving at-least-once delivery in a Storm topology, showing the interaction between spouts, bolts, and the AckerBolt.

- **Spout:** The spout is the source of streams in a Storm topology. It emits tuples to be processed by bolts.
- **Bolts:** Bolts process the incoming tuples from spouts or other bolts. They can emit tuples to other bolts downstream.
- **Acker Bolts:** Storm uses acker bolts to track the processing of tuples. Each emitted tuple is assigned a unique ID, and acker bolts keep track of these IDs to ensure that all tuples are processed at least once.

To ensure at-least-once processing, Storm employs the following mechanisms:

1. **Tuple Anchoring:** Each tuple emitted by a spout is anchored. Bolts process these tuples and emit new ones, maintaining the anchor to the original tuple.
2. **Acknowledgement and Failure Tracking:** When a bolt finishes processing a tuple, it acknowledges the tuple. If a tuple is not acknowledged within a certain timeout, it is considered failed and re-emitted.
3. **Acker Bolts:** These bolts track the lineage of tuples and ensure that all tuples derived from the original spout tuple are processed. If any part of the processing fails, the acker bolts ensure the tuple is re-emitted and reprocessed.

The architecture, as depicted in the figures ensures that all tuples are processed at least once by tracking their processing and re-emitting them if necessary. This robust mechanism provides reliable data processing in a distributed system.

Apache Flink

Apache Flink is a high-performance, reliable, and scalable data stream processing framework, presently among the most popular choices for such applications.

- Initially developed at TU Berlin in 2011, it has incorporated numerous ideas from database systems, such as operators and transactional mechanisms.
- Key characteristics of Flink include its capability for both stream-processing and batch-processing, support for complex event processing, and its low latency and high throughput performance.
- It provides exactly-once processing guarantees and supports a wide array of APIs, which has led to its adoption by companies like Uber, eBay, and Zalando.

Anatomy of a Flink Cluster

The following figure illustrates the architecture of a Flink cluster, delineating the roles of the Task Managers and the Job Manager in processing a data flow graph.

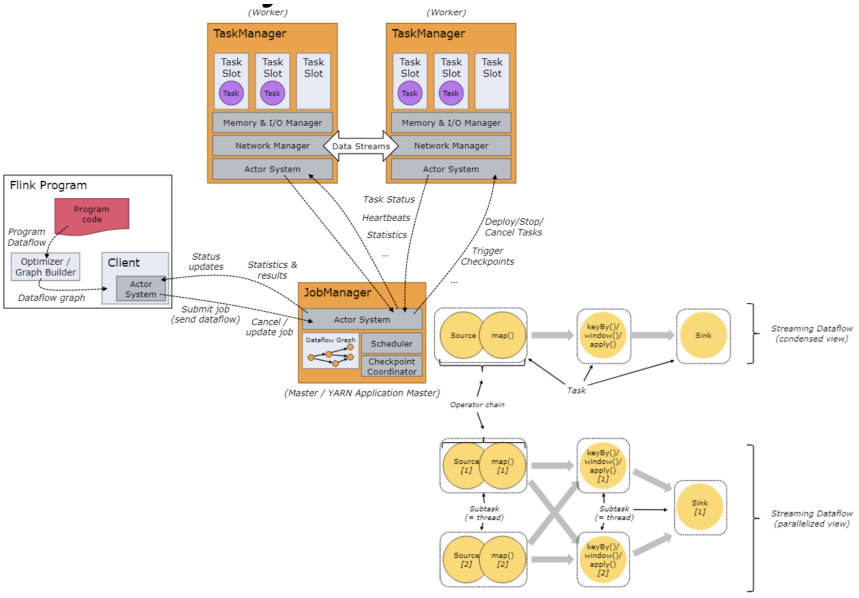


Figure 58: The detailed structure of a Flink cluster, showing the interaction between the client, the Job Manager, and the Task Managers.

Spark Streaming (Old API)

Spark Streaming's original processing model is based on micro-batch computations, where the input data stream is divided into batches that are processed by the Spark Engine to generate corresponding batches of processed data.

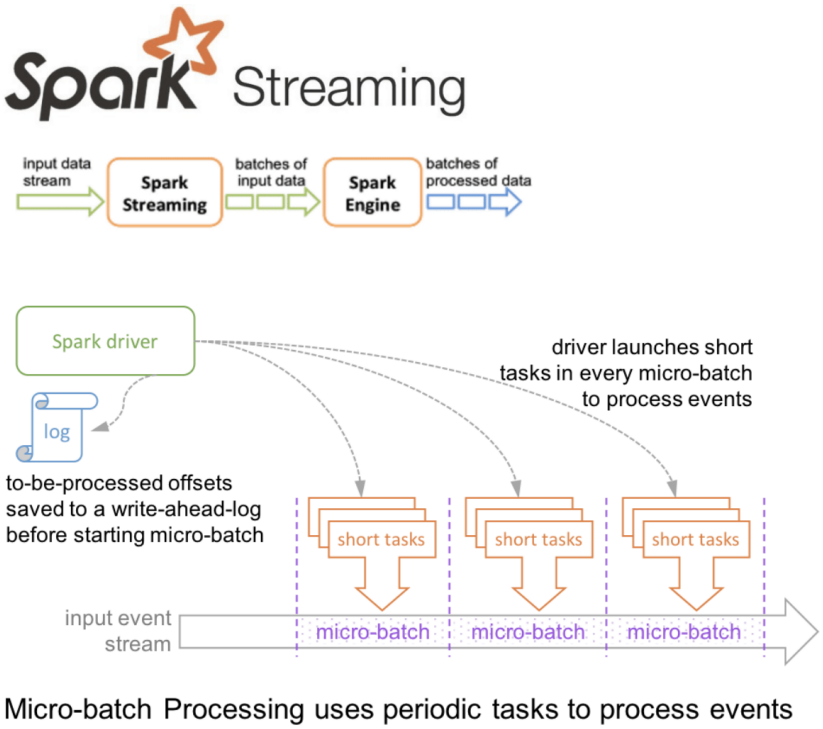


Figure 59: The architecture of Spark Streaming using the old API, showcasing micro-batch processing and the role of the Spark driver.

In this model, the Spark driver orchestrates the processing by launching tasks for each micro-batch to ensure that events are processed effectively. Fault tolerance is maintained by recording offsets in a write-ahead log before processing begins.

Spark Structured Streaming

Spark Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. It treats streams of data as unbounded tables and enables rich, interactive queries on streaming data.

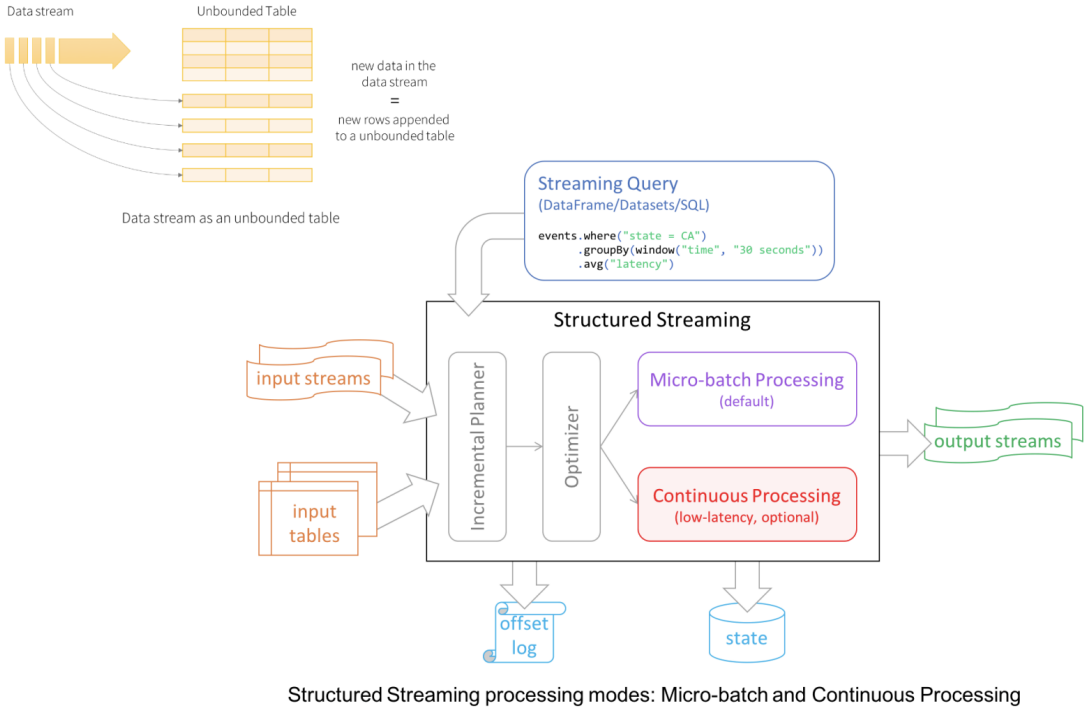


Figure 60: The architecture of Spark Structured Streaming, showing how data streams are treated as unbounded tables and processed using either micro-batch or continuous processing modes.

The system is designed to process streaming data in either a micro-batch mode, which is the default processing mode, or an optional continuous processing mode for lower-latency requirements. It maintains a state and offset log to track the progress and to ensure fault tolerance.

The key components and process flow described in the image are:

- Data Stream: The raw, unbounded sequence of data records that continuously flow into the system.
- Unbounded Table: The stream is interpreted as an unbounded table where new data equates to new rows being appended.
- Input Streams and Input Tables: Streams are converted into tables for processing.
- Streaming Query: A query performed on the data using DataFrame/Dataset/SQL interfaces in Spark, with the capability to handle streaming data.
- Structured Streaming: The overall framework that allows querying streaming data using the same APIs that are used for batch processing.
- Micro-batch Processing (default): The default processing mode in Structured Streaming, where data is processed in small batches.

- Continuous Processing (low-latency, optional): An optional mode that allows for lower latency by processing records as they arrive rather than in micro-batches.
- Offset Log: Tracks the progress of which data has been processed.
- State: Holds intermediate state information for aggregations or window operations.
- Output Streams: The result of the streaming computation, equivalent to the modified rows in the unbounded table.

This framework allows for complex streaming analytics, combining the ease of batch processing with the power and immediacy of streaming processing.

Comparison: Storm, Spark and Flink

Figure 61 provides a comparison of the main features of Storm, Spark, and Flink.

Feature	Storm	Spark	Flink
Processing Model	Event-Streaming	Micro-Batching / Batch (Spark Core)	Event-Streaming and Batch
Delivery Guarantees	At most once / At least once	Exactly Once	Exactly Once / At Least Once
Latency	Sub-second	Seconds	Sub-second
Language Options	Java, Clojure, Scala, Python, Ruby	Java, Scala, Python	Java, Scala, Python
State Management	Limited, no built-in support	Uses external storage	Built-in state management
Windowing	Limited support	Time-based, processing-time windows	Advanced time and event-time windows
Fault Tolerance	Tuple tracking and acking	RDD lineage and checkpointing	Stateful stream processing with exactly once consistency
Development	Use other tools for batch	Batching and streaming are very similar	Integrated support for batch and streaming

Figure 61: Comparison of Storm, Spark, and Flink

General Discussion

Processing Model:

- **Storm** primarily focuses on real-time stream processing using an event-streaming model. It processes each event as it arrives.

- **Spark** employs a micro-batching approach where streams are divided into small batches for processing, providing a unified engine for both batch and stream processing.
- **Flink** supports both event-streaming and batch processing. It treats batch jobs as a special case of streaming, providing low-latency event processing.

Delivery Guarantees:

- **Storm** offers at-most-once and at-least-once guarantees through tuple tracking and acking mechanisms.
- **Spark** ensures exactly-once processing semantics by using its built-in fault tolerance and RDD lineage.
- **Flink** provides exactly-once and at-least-once processing guarantees using stateful stream processing and checkpointing.

Latency:

- **Storm** is designed for low-latency, sub-second event processing, making it suitable for real-time applications.
- **Spark** has higher latency due to its micro-batching nature, typically processing data within seconds.
- **Flink** offers low-latency processing, comparable to Storm, by leveraging event-driven processing.

Language Options:

- **Storm** supports Java, Clojure, Scala, Python, and Ruby.
- **Spark** supports Java, Scala, and Python.
- **Flink** supports Java, Scala, and Python, with a strong focus on Java and Scala.

State Management:

- **Storm** has limited built-in support for state management and relies on external systems for maintaining state.
- **Spark** uses external storage systems for state management, which can add latency.
- **Flink** includes built-in, robust state management capabilities that are tightly integrated with its stream processing model, providing efficient state handling and recovery.

Windowing:

- **Storm** has limited support for windowing operations.
- **Spark** provides time-based and processing-time windowing capabilities.
- **Flink** offers advanced windowing operations, including time and event-time windows, which are crucial for accurate event-time processing in stream applications.

Fault Tolerance:

- **Storm** achieves fault tolerance through tuple tracking and acknowledgment, resending tuples if processing fails.
- **Spark** uses RDD lineage and checkpointing to recover from faults and ensure data integrity.
- **Flink** uses stateful stream processing with exactly-once consistency and checkpointing to maintain fault tolerance and state recovery.

Development:

- **Storm** requires separate tools for batch processing, focusing mainly on streaming.
- **Spark** provides a unified engine for both batch and streaming, making development consistent across both paradigms.
- **Flink** seamlessly integrates batch and stream processing, treating batch processing as a special case of streaming.

In summary, Storm is tailored for real-time stream processing with low latency, Spark offers a unified approach to batch and micro-batch stream processing, and Flink combines the strengths of both with robust state management and low-latency event streaming. Each framework has its unique features and advantages, making them suitable for different types of data processing workloads.