Big Data Processing

Summary of All the Lecture

Introduction

- There are V's associated with Big Data
  - **Volume** → large amounts of data
  - **Variety** → different forms and diverse sources
    - Structured data → sql tables, well-known formats
    - Semi-structured data → JSON, XML
    - Unstructured data → Text, audio, video
    - These types of sources are usually combined
  - **Velocity** → changing quickly
    - Not only the data is big, but the timeliness is also really big
    - Data needs to be processed with soft or hard real-time guarantees
  - Many more like Value, Validity, Veracity (trustworthiness), Volatility, Visiblity, Virality
- In general BDP goes through the **ETL lifecycle**
  - Extract: convert raw or semi-structured data into structured data
  - Transform: convert units, join data sources, cleanup
  - Load: load the data into another system for further processing
  - BD Engineering → building pipelines
  - BD analytics → discovering patterns
- There are two ways to process the data
  - **Batch Processing**
    - data exists in some data store
    - program processes the whole data at once
  - **Stream Processing**
    - Process data as it arrives at the system
- Two basic approaches to distribute data on several machines
  - **Data Parallelism**
    - divide data in chunks
    - apply the same algorithm to the chunks
  - **Task Parallelism**
    - divide the problem in chunks
    - run it on a cluster of machines

- We want our BDP systems to have
  - Robustness and fault-tolerance
  - Low latency reads and updates
  - Scalability
  - Generalization
  - Extensibility
  - Ad hoc queries
  - Minimal maintenance
  - Debuggability

Scala

- **Scala Overview**
  - Compiled to JVM bytecode
  - Can interoperate with JVM libraries
  - Does not depend on indentation for blocks (uses `{}` to denote blocks)
- **Type Inference**
  - Scala uses type inference extensively
    - `val` is used for immutable variables (single-assignment)
    - `var` is used for mutable variables (can be reassigned)
    - Example:

      ```
      val a: Int = 5    // explicit typing
      val b = 10        // type inferred
      var x = 20        // mutable variable
      x = 30            // re-assignment allowed
      ```
- **Function Declaration**
  - Functions are statically typed, and their return types are inferred
  - Example:

    ```
    def max(x: Int, y: Int): Int = {
      if (x >= y) x else y
    }
    ```
- **Higher-Order Functions**
  - Scala supports higher-order functions (functions that take other functions as arguments)
  - Example:

    ```
    def bigger(x: Int, y: Int, f: (Int, Int) => Boolean) = f(x, y)
    bigger(1, 2, (x, y) => x > y) // returns false
    ```

- **Object-Oriented Programming**
  - Scala supports both functional and object-oriented programming paradigms
  - Classes in Scala:

    ```
    class Foo(val x: Int, var y: Double)
    val obj = new Foo(1, 2.0)
    println(obj.x)  // read-only
    obj.y = 3.0     // mutable
    ```
- **Pattern Matching**
  - Pattern matching in Scala is a powerful feature similar to `if..else` but more versatile
  - Example:

    ```
    val value = 5
    value match {
      case 1 => "One"
      case x if x > 2 => "Greater than two"
      case _ => "Other"
    }
    ```

Functional Programming

- **Key Concepts**
  - **Absence of Side-Effects**
    - Pure functions return the same output for the same input without modifying external state
  - **Immutable Data Structures**
    - Data structures in functional programming are immutable, meaning they cannot be changed after they are created
    - Example: Immutable List in Scala:

      ```
      val nums = List(1, 2, 3)
      val newNums = 4 :: nums  // new list, original unchanged
      ```
  - **Higher-Order Functions**
    - Functions that accept other functions as parameters or return functions as results
    - Examples of higher-order functions: `map`, `filter`, `fold`

      ```
      val nums = List(1, 2, 3, 4)
      val squared = nums.map(x => x * x) // List(1, 4, 9, 16)
      ```
  - **Laziness**
    - Lazy evaluation delays the computation of expressions until their values are needed
    - Example:

      ```
      lazy val x = { println("Evaluating x"); 10 }
      ```

```
println(x)  // prints "Evaluating x" only when accessed
```

- **Pure Functions**
    - A function that depends only on its input parameters and does not have any observable side-effects
    - Referential transparency allows replacing an expression with its value without changing program behavior
- **Recursion**
    - Functions often use recursion in functional programming, replacing traditional loops
    - Example:

    ```
    def factorial(n: Int): Int = {
      if (n <= 1) 1
      else n * factorial(n - 1)
    }
    ```
- **Monads**
    - Monads are structures that represent computations
    - Monads are a design pattern that define how function can be used togetehr to build generic types
    - A value-wrapping type that:
        - has an identity function
        - Has a `flaMap` function
    - Monads used to deal with side-effects in FP
        - NPE → Option[T]
        - Exceptions → Try[T]
        - Latency in Async → Future[T]
    - Example in Scala with `Option` (dealing with null values):

    ```
    def getValue(input: Int): Option[Int] = if (input > 0) Some(input) else No
    ne
    val result = getValue(5).map(_ * 2)  // result is Some(10)
    ```
- **Immutability**
    - Immutability ensures that once a data structure is created, it cannot be altered
    - Immutable data structures allow safe concurrent access without locks

    ```
    val nums = List(1, 2, 3)
    val updatedNums = nums :+ 4  // new list created with added element
    ```
- **Pattern Matching in Functional Context**
    - Can deconstruct data structures using pattern matching
    - Example with recursive list matching:

```
def length(lst: List[Int]): Int = lst match {
  case Nil => 0
  case _ :: tail => 1 + length(tail)
}
```

- **Common Functional Constructs**
  - **Map**: Applies a function to all elements in a collection

    ```
    val nums = List(1, 2, 3)
    val doubled = nums.map(_ * 2)  // List(2, 4, 6)
    ```

  - **Fold**: Reduces a collection to a single value

    ```
    val sum = nums.foldLeft(0)(_ + _)  // sum of list elements
    ```

  - **FlatMap**: Like map, but flattens the result

    ```
    val list = List(List(1, 2), List(3, 4))
    val flat = list.flatMap(x => x)  // List(1, 2, 3, 4)
    ```

Immutability

- **Definition**
  - Immutability means that once a data structure is created, it cannot be changed. Instead of modifying an object, a new object is created.
  - Immutable data structures are used extensively in functional programming due to their ability to support side-effect-free operations.
- **Advantages of Immutability**
  - **Concurrency**: Immutable objects are inherently thread-safe since they cannot be modified after creation, reducing the need for locks in concurrent programming.
  - **Predictability**: Functions working with immutable data structures are easier to reason about since the state doesn't change unexpectedly.
  - **Referential Transparency**: Expressions can be replaced with their values without changing the program's behavior.
- **Immutable Data Structures in Scala**
  - Scala provides both mutable and immutable collections. When in doubt, prefer the immutable version.
  - Examples:

    ```
    val nums = List(1, 2, 3)          // immutable list
    val newNums = 0 :: nums           // creates a new list with '0' at the front
    val updatedNums = nums :+ 4       // creates a new list with '4' appended
    ```

  - Immutable collections do not allow in-place modification, making them ideal for functional programming.

Copy-On-Write (COW)

- **Definition**
  - Copy-On-Write (COW) is an optimization strategy where copies of objects are not immediately created upon modification. Instead, multiple processes share the same data in memory until one process attempts to modify it, triggering the creation of a new, private copy.
- **How it Works**
  - When a write operation is required, the system makes a copy of the original data, and the write is performed on the copied version. The original version remains intact for other processes that do not require modification.
  - This technique is common in multi-threaded environments and systems where read operations vastly outnumber write operations.
- **Advantages of COW**
  - **Memory Efficiency**: Processes share the same memory until a write operation occurs, reducing unnecessary duplication of data.
  - **Concurrency**: Multiple processes can read the same data concurrently without locking, improving performance in read-heavy systems.
  - **Fault Isolation**: When a process modifies a resource, only its local copy is affected, preventing unintended side effects in other processes.
- **Examples of COW in Systems**
  - **Operating Systems**: COW is used in OS-level process management (e.g., `fork` system call), where parent and child processes initially share memory pages until one of them writes to a page, at which point it is copied.
  - **File Systems**: Modern file systems like BTRFS and ZFS use COW to efficiently handle snapshots and versioning.
- **Example of COW in Scala (Immutable Data Structures)**
  - Even though immutable data structures look like they are copied on each operation, they internally reuse memory whenever possible, applying COW principles.

    ```
    val tree = TreeSet[Char]('a', 'b', 'c')
    val newTree = tree + 'd'  // COW creates a new version, but reuses existin
    g structure
    ```

Unix Fundamentals

- **Unix Overview**
  - A true multiuser operating system where multiple users can run multiple processes.
  - Users have their own space on the machine and are identified by an ID number.
  - The root user (`id 0`) has superuser privileges and controls the system.
  - Everything in Unix is represented as a file: configuration files, devices, etc.
- **Filesystem Structure**
  - Unix filesystem has specific directories:
    - `/bin`: Essential binaries.
    - `/etc`: Configuration files.

        - `/home`: User home directories.
        - `/lib`: Libraries for the system.
        - `/tmp`: Temporary files.
        - `/usr`: Additional programs.
    - Permissions are used to control access to files:
        - `r` (read), `w` (write), and `x` (execute).
        - Example:

```
$ ls -la /bin/ls
-rwxr-xr-x 1 root wheel 38624 Jul 15 06:29 /bin/ls
```

- **Path Types**
    - **Absolute Paths**: Start from root `/`, e.g., `/var/log/messages`.
    - **Relative Paths**: Start from the current directory, e.g., `./log/messages`.

File Manipulation Commands

- **File Listing**
    - `ls` lists files:
        - `l`: List details.
        - `a`: Include hidden files.
- **File Creation/Manipulation**
    - `touch <file>`: Create a new file or update its timestamp.
    - `cp <from> <to>`: Copy a file or directory.
        - `R`: Recursive copy.
        - `p`: Preserve permissions.
    - `mv <from> <to>`: Move or rename files.
    - `mkdir <dir>`: Create directories.
        - `p`: Create intermediate directories if needed.

Text File Processing

- **Viewing Files**
    - `cat <file>`: Display file content.
    - `less <file>`: View file interactively, scroll up/down with arrow keys.
    - `head <file>`, `tail <file>`: Display the first/last lines.
        - `n`: Specify the number of lines.
- **Searching Files**
    - `grep <pattern> <file>`: Search for a pattern in a file.
        - `i`: Case insensitive.
        - `v`: Invert the match.
        - `n`: Show line numbers.

        - `R`: Recursive search.
- **Pattern Matching**
    - ` `: Matches any character(s).
    - `.`: Matches any single character.
    - `^`: Beginning of line.
    - `$`: End of line.
    - `[a-z]`: Range of characters.

Process Management

- **Managing Processes**
    - Unix allows multitasking, running multiple processes simultaneously.
    - `ps`: Display running processes.
        - Example output:

          ```
          UID   PID  PPID   C STIME   TTY       TIME CMD
          0     1     0     0  28Nov17 ??       21:20.80 /sbin/launchd
          ```

    - `kill -<signal> <pid>`: Send a signal to a process.
        - `TERM`: Graceful termination.
        - `KILL`: Forceful termination.
- **Background Processes**
    - Append `&` to a command to run it in the background.
        - Example: `find / | sort &`

Text Flow Control

- **Redirection**
    - Redirect output to a file using `>`, and append using `>>`.
        - Example:

          ```
          ps -ef > processes.txt
          ```

    - Redirect input from a file using `<`.
- **Piping**
    - Use `|` to pipe the output of one command as the input to another.
        - Example: `cat file.txt | grep "pattern"`

Advanced Text Processing

- **Sort, Cut, and Uniq**
    - `sort`: Sort lines of text.
        - `n`: Numeric sort.
        - `r`: Reverse sort.

- Example:

  ```
  sort -n -k2,2 -t ',' data.csv
  ```

- `cut`: Extract specific columns from a file.
  - Example:

    ```
    cut -f1,6 -d: /etc/passwd
    ```

- `uniq`: Find unique lines in a sorted file.
  - Example:

    ```
    sort file.txt | uniq
    ```

- **Stream Processing with Sed and AWK**
  - **Sed**: Stream editor for modifying files.
    - Example:

      ```
      sed -e 's/foo/bar/' < file.txt
      ```

    - Deletes lines:

      ```
      sed -e '3d' -e '5d' < file.txt
      ```

  - **AWK**: A text processing language.
    - Example:

      ```
      awk '{print $2}' licenses.txt
      ```

Pipelines and Task Automation

- **Pipelines**
  - Chain commands together to create powerful pipelines.
    - Example:

      ```
      cat access_log | grep "foo" | tail -n 10
      ```

  - `xargs`: Execute a command on each input line.
    - Example:

      ```
      find . -type f | xargs wc -l
      ```

- **Make**
  - `make` is used to automate the execution of commands based on dependencies.
  - Example Makefile:

```
all: output.txt
output.txt: input.txt
  cat input.txt > output.txt
```

Networking and SSH

- **SSH**
  - Securely connect to remote systems and run commands.
    - Example:

      ```
      ssh user@host "ls /home/user"
      ```

  - **Tunneling** with SSH:
    - Example:

      ```
      ssh -L 8080:localhost:8080 user@remote
      ```

- **Curl**
  - `curl` fetches data from URLs.
    - Example:

      ```
      curl -s <https://api.example.com/data> | jq .
      ```

Command Line Utilities for Data Processing

- **Basic Utilities**
  - `wc`: Count lines, words, and characters.
  - `tee`: Split output and write to multiple locations.
  - `tr`: Translate characters (e.g., uppercase to lowercase).
  - Example:

    ```
    tr 'A-Z' 'a-z' < file.txt
    ```

- **Advanced Filtering**
  - `jq`: A JSON processor.
    - Example:

      ```
      curl -s <https://api.example.com/data> | jq '.results'
      ```

- **Scheduling with Cron**
  - Cron jobs allow periodic execution of commands.
    - Example cron entry (run every day at 1 am):

      ```
      0 1 * * * /path/to/script.sh
      ```

Version Control

- **Git**
    - `git log`: View the history of commits.
        - Example:

          ```
          git log --oneline --graph
          ```
    - `git blame`: View line-by-line commit history.
        - Example:

          ```
          git blame file.txt
          ```
    - `git diff`: Compare changes between two commits or files.
        - Example:

          ```
          git diff HEAD~1 file.txt
          ```

Writing Bash Scripts

- **Bash Script Basics**
    - A bash script is a plain text file containing a series of commands that can be executed in sequence.
    - To write a bash script:
        1. Create a file and give it the `.sh` extension (not mandatory but a good practice).
        2. Add the shebang at the top of the file to indicate the interpreter:

           ```
           #!/bin/bash
           ```
        3. Make the script executable:

           ```
           chmod +x script.s
           ```
        4. Run the script:

           ```
           ./script.sh
           ```
- **Variables**
    - Variables store data for use within the script.
    - Variables are created by assigning a value without spaces around the `=` sign.
    - Example:

      ```
      greeting="Hello, World!"
      echo $greeting
      ```
    - You can also pass arguments to the script and access them using `$1`, `$2`, etc. `$0` refers to the script name.

        - Example:

          ```
          #!/bin/bash
          echo "First argument: $1"
          echo "Second argument: $2"
          ```

- **Control Structures**
    - **Conditionals**:
        - Use `if`, `elif`, and `else` for conditional logic.
        - Example:

          ```
          if [ $1 -gt 10 ]; then
            echo "Number is greater than 10"
          else
            echo "Number is less than or equal to 10"
          fi
          ```

    - **Loops**:
        - **For loop**: Iterate over a list or range.

          ```
          for i in 1 2 3; do
            echo "Item: $i"
          done
          ```

        - **While loop**: Execute commands as long as a condition is true.

          ```
          count=0
          while [ $count -lt 5 ]; do
            echo "Count: $count"
            count=$((count + 1))
          done
          ```

- **Functions**
    - Functions allow you to reuse code within your script.
    - Example:

      ```
      greet() {
        echo "Hello, $1!"
      }
      greet "Alice"
      ```

    - Functions are invoked by simply calling their name followed by arguments (if any).
- **Error Handling**
    - You can check the exit status of a command using `$?`.
    - If the exit status is 0, the command was successful; otherwise, an error occurred.
        - Example:

```
mkdir newdir
if [ $? -eq 0 ]; then
  echo "Directory created successfully."
else
  echo "Failed to create directory."
fi
```

    - Use `set -e` to stop the script if any command fails.

- **Input and Output**
    - **Reading User Input**: Use `read` to capture user input.
        - Example:

```
echo "Enter your name:"
read name
echo "Hello, $name!"
```

    - **Redirecting Output**:
        - `>`: Redirect output to a file.

```
echo "Hello" > output.txt
```

        - `>>`: Append output to a file.

```
echo "World" >> output.txt
```

- **Command Line Arguments**
    - Arguments can be passed when calling the script and accessed via `$1`, `$2`, etc.
    - `$#`: Number of arguments passed.
    - `$@`: All arguments passed to the script.
    - Example:

```
#!/bin/bash
echo "Total arguments: $#"
echo "All arguments: $@"
```

- **Special Operators**
    - **Logical Operators**:
        - `&&`: Run the second command only if the first succeeds.

```
mkdir newdir && echo "Directory created."
```

        - `||`: Run the second command only if the first fails.

```
mkdir existingdir || echo "Directory already exists."
```

Distributed Systems

- What is a Distributed System?
    - Software system in which components located on networked computers communicate and coordinate their actions by passing messages.
- Parallel vs. Distributed Systems
    - Parallel Systems → use shared memory
        - Distributed Parallel systems → use shared memory
            - coordinated with special HW/SW that unifies memory accesses across multiple computers
    - Distributed Systems → use no shared components.
- Why Distributed Systems
    - Inherent Distribution
        - Information dissemination (publishers and subscribers)
        - Distributed process control
        - Cooperative work (different nodes on a network read/write)
        - Distributed storage
    - Distribution as an Artifact
        - Performance
        - Scalability
            - No longer really Moore's law → distribution helped with advancement
        - Availability
        - Fault tolerance
- What should a Distributed System have?
    - Computational entity each with own memory
        - They need to synchronize their states
    - Communication through message passing
    - Each entity part of bigger picture
    - Need to tolerate failure
- Disadvantages of Distributed Systems
    - They fail often (hard to spot)
        - split-brain scenarios
    - Maintaining order/consistency is hard
    - Coordination is hard
    - Partial operation must be possible
    - Testing is hard
    - Profiling is hard
- Fallacies of Distributed Systems
    - The Network is Reliable
    - Latency is zero

    - Bandwidth is infinite
    - The network is secure
    - Topology does not change
    - Transport cost is zero
    - The network is homogeneous
- Four Main issues
    - Partial Failures
        - Any one point in the system can fail
        - Systems must continue to function correctly as a whole
        - Hard to detect whether something failed or not
    - Unreliable Networks
        - Async vs Sync Systems
            - There are two types of network systems.
            - Synchronous Systems
                - Process Execution or message delivery times are bounded
                - purely sync systems exist only in theory
                - In these types of systems, we have
                    - Timed failure detection
                    - Time based coordination
                    - Worst-case performance
            - Asynchronous Systems
                - No assumptions about timing is made
                - used by most distributed system for some form of communication over a network
                - There are various points of failure in an Async system
                    - The request can get lost
                    - The remote node can be down
                    - The response can get lost
                - Usually timeouts work with the addition of retransmission until success
                - retransmission of the message may help ensure the reliability of the communication links but they can also slow down the system
                    - Systems have to still have some notion of Syncness
                - Partially-Synchronous Systems
                    - There exist upper bounds on the network delay but the programmer does not know them.
    - Unreliable time
        - Time is the only global constant nodes can rely on to make distributed decision on ordering problems
            - Examples of where time is necessary

        - Sequencing items in memory
        - Mutual exclusion of access to a shared resource
        - Encoding history
        - Transactions in a database
        - Consistency of distributed data
        - Debugging
- Two ways of keeping track of time in computers
    - Real time clocks (RTCs) → Capture our intuition about time and are kept in sync with the NTP protocol with centralized servers
    - Monotonic Clocks → Those clocks only move forward
- Time in Centralized vs. Distributed Systems
    - Centralized Systems
        - System Calls to Kernel
        - Monotonically increasing time values
    - Distributed Systems
        - Achieving agreement on time is not trivial.
- Trouble With Computer Clocks
    - Monotonic clocks are maintained by the OS and rely on HW counters exposed by the CPUs
        - Each node has it's own notion of time
        - NTP can synchronize time across nodes with an accuracy of ms.
        - leap seconds are introduced → minute taking 59 or 61 seconds
        - microsecond accuracy with GPS clocks
- Logical Time
    - Instead of using precise clock time, capture the events relationship between a pair of events
    - Base on causality
        - If some event possibly causes another event → first event happened before the other
    - Here we have the notion of Order
        - A way of arranging items in a set so that the following properties are maintained
            - Strict Partial Order
                - **Irreflexivity** : $\forall a. \neg a < a$ (items not comparable with self)
                - **Transitivity**: if $a \leq b$ and $b \leq c$ then $a \leq c$
                - **Antisymmetry**: if $a \leq b$ and $b \leq a$ $a = b$
            - Strict Total Order
                - An additional property: $\forall a, b$, $a \leq b \lor b \leq a \lor a = b$
        - Relation of order to Time
            - FIFO is enough to maintain order with a single sender

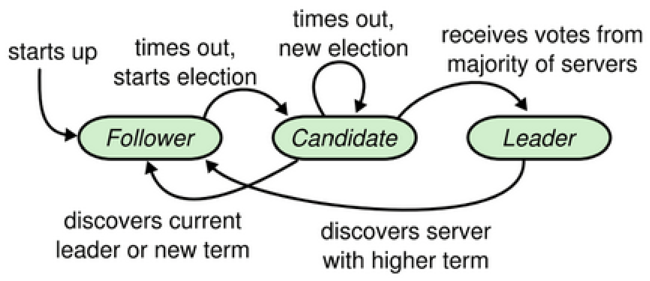
        - Time at the receiver end is enough to maintain at the receiver end
        - With multiple senders and receivers, we need another ordering scheme:
            - Total Order: If message rate is globally bounded, and less fine-grained than our clock accuracy, the synchronized RTCs are enough
            - Causal Order: Othersie, we need to rely on happens before relationships
    - Happens-before relation
        - Events in distributed systems
            - A process performs some local computation
            - A process sends a message
            - A process receives a message
        - What does Lamports happens-before say
            - If a and b are events in the same node, and a occurs before b, then a → b
            - if a is the event of sending a message and b is the event of receiving that message, then a → b
            - The relation is transitive
        - Two events not related to happened before are concurrent
    - The Idea of Lamport timestamps
        - each individual process p matains a counter $LT(p)$
        - when a process p performans an action, it increments $LT(p)$
        - When a process p sends a message, it includes $LT(p)$ in the message.
        - When a process p receives a message from a process q, that message includes the value of $LT(q)$; p updates its $LT(p)$ to the $max(LT(p), LT(q)) + 1$
        - Therefore we can say for two events a and b with relation a → b, holds that $LT(a) < LT(b)$
    - The also exists the notion of Vector Clocks
        - These clocks can maintain causal order
        - On a system with N nodes, each node i maintains a vector Vi of size N
            - Vi[i] is the number of events that occurred at node i
            - Vi[j] is the number of events that node i knows occurred at node j
        - Values in these vectors are updated as follows
            - Local events increment Vi[i]
            - When i sends a message to j, it includes Vi
            - when j receives Vi, it updates al elements of Vj to the max of each index in the vector Vi and Vj
        - We can compare to vectors Vi and Vj
            - Vi = Vj iff Vi[k] = Vj[k] for all k
            - Vi ≤ Vj iff Vi[k] ≤ Vj[k] for all k
            - Concurrency Vi || Vj otherwise

        - Vector clocks give us some guarantees
            - if a → b then VC(a) < VC(b)
            - if VC(a) < VC(b), then a → b
        - Vector clocks are, however, expensive to maintain
        - They require O(n) timestamps to be exchange with each communication
        - They are the best we can do
- No single source of truth, different opinions
    - What is true in a distributed setting?
        - Nodes in distributed system cannot know anything for sure
        - Individual nodes cannot rely on their own information
        - Split-brain scenarios can exist in which parts of the systems know a different version of the truth than other parts
        - Reaching consensus is a fundamental problem in distributed systems.
    - Consensus for Distributed Decision Making
        - Providing agreement in the presence of faulty nodes
            - Resource allocation
            - Committing a transaction
            - Synchronizing state machines
            - Leader Election
            - Atomic Broadcasts
    - 2-generals problems
        - The two generals problem setting
            - 2 armies camped in opposing hills (A1 and A2)
            - The are only able to communicate with messengers
            - They need to decide on a time to attack
            - Enemy (B) is camped between the two hills and can at any time intercept the messengers
        - Approximate solutions
            - Pre-agree on timeouts
            - send n labeled messages
            - receiver calculates received messages within time window, then decides how many messages to send for ack
    - FLP Impossiblity
        - *"It is impossible to have a deterministic consensus algorithm that can satisfy agreement, validity, termination, and fault tolerance (of even a single process) in an asynchronous distributed system. [FLP'85]*
          "

        - So the FLP theorem says, In an async network, consensus cannot be reached if at least one node fails in async networks
        - The results show that distributed consensus is impossible. They assume:
            - Async Communication
            - No clocks or timeouts
            - No random number generators
        - In an async system there is no algorithm that solves consensus in every possible run
        - We can mitigate the consequences using randomization and partial synchrony
    - Byzantine Generals Problem
        - the Byzantine generals problem shaped distributed systems research for the next 40 years.
        - Problem Formulation
            - Several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general.
            - They must decide upon a common plan of action: Attack or Retreat.
            - The generals can communicate with each other only by messengers.
            - There might be traitors (malicious or arbitrary behavior).
            - All loyal generals must agree on a plan.
        - Solution → Fault tolerant consensus
            - With only three generals no solution can work in the presence of a single traitor
            - fault tolerant consensus
                - PBFT → Byzantine fault tolerant consensus with at least 3f + 1 nodes with f trators
                - Paxos and Raft → crast fault tolerant consensus with at least 2f + 1 nodes with f faulty nodes
    - Consensus Protocols
        - What is a concensus protocol?
            - defines a set of rules for message exchange and processing for distributed components to reach agreement
            - Basic properties of crash fault-tolerant consensus protocols include
                - Safety: Never returning an incorrect result, in the presence of non- Byzantine conditions
                - Availability: Able to provide an answer if n/2 + 1 servers are operational
                - No Clocks: they do not depend on RTCs to work
                - Immune to Stranglers: if n/2 + 1 server vote, then the result is considered safe.
        - State Machine Replication (SMR)
            - can be used to keep the log module of the replicated state machines in sync

            - consensus protocol provides all replicates to agree on the same order of commands/entries
            - Paxos and Raft are crash fault tolerant consensus protocols for SMR
- Paxos consensus algorithm
    - Roles
        - Proposer → chooses a value (or receives from a client) and sends it to a set of accepters to collect votes
        - Acceptor → Vote to accept or reject the values proposed by the proposer. For fault tolerance, the algorithm requires only a majority of acceptor votes
        - Learner → They adopt the value when a large enough number of acceptors have accepted it.
    - Every proposal consists of a value, proposed by the client and unique monotonically increasing proposal number, aka ballot number.
    - Acceptance of the proposals by a majority of processes/servers provide fault tolerance.
    - Steps
        - Phase 1 - Voting
            - Prepare
                - A proposer selects a proposal number n and sends a “prepare” request Prepare(n) to acceptors.
            - Promise
                - If n is higher than every previous proposal number received, the acceptor returns “Promise”
                    - aka it will ignore all future proposals having number less than n
                - If the acceptor accepted a proposal, it must include the previous number, m, and the corresponding accepted value, w, in its response tot he proposer
        - Phase 2 - Replication
            - Accept
                - If the proposer receives a response from a majority of acceptors, then it sends an accept request for a proposal numbered n with the highest-numbered proposal among the responses
            - Accepted
                - If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater or equal than the proposal number.
- Raft Consensus Algorithm
    - Leader-based asymmetric model
        - A node in a system can only be in one of the three states at any point in time
            - Leader
            - Follower
            - Candidate

- Separates Leader Election and Log Replication States


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

- Cluster states
    - Leader Election
        - Select one server to act as leader
        - Detect crashes, choose new leader
        - Process
            - Raft defines the following server states
                - Candidate → candidate for being a leader, asking for votes
                - Leader → accepts log entries from clients, replicates them on other servers
                - Follower → replicate the leader's state machine
            - Each server maintains current term value
                - Exchanged in every RPC
                - Peer has later term? update term, revert to follower
                - Raft selects one leader at each term
                    - Some terms the election can fail and result in no leader for that term
    - Log Replication
        - leader accepts commands from clients, appends to its log
        - leader replicates its log to other servers (overwrites inconsistencies)
        - Each log entry also has an integer index identifying its position in the log
        - The leader sends `AppendEntries` message to append an entory the log
        - A log entry is committed once the leader that created the entry has replicated it on a majority of the servers.
- Raft ensures that
    - logs are always consistent and that only servers with up-to-date logs can become leaders

- Byzantine fault tolerant consensus
    - Raft and Paxos cannot deal with Byzantine circumstances
    - in open distributed systems, this assumption is not valid
    - decision require majority votes, typically require $n \geq 3f + 1$ nodes where f is the number of faulty nodes

        - Practical Byzantine Fault Tolerance (PBFT) and BitCoin examples of Byzantine fault tolerant consensus algorithms

Distributed Databases

- What is a distributed database?
    - A distributed system designed to provide read/write access to data, using a specified data format
- Splitting data among nodes
    - Replication
        - Keep a copy of the same data on several different nodes
        - Why?
            - we keep identical copies of data on different nodes
            - It helps with
                - Data Scalability
                    - To increase read throughput by allowing more machines to serve read-only requests
                - Geo-scalability
                    - To have the data close to the clients
                - Fault tolerance
                    - To allow the system to work even if parts are down
        - How?
            - Replication Architectures
                - In a replicated system, we have two node roles
                    - Leaders or Masters → Nodes that accept write queries from clients
                    - Followers, Slaves or replicas → Nodes that provide read-only access to data
                - Leader-based
                    - Single Leader / Master-Slave
                    - A single leader accepts writes, which are distributed to followers
                - Multi-leader
                    - Master-master
                    - multiple leaders accept writes, keep themselves in sync, then update followers
                    - complications
                        - Multiple leader nodes to accept writes
                        - replication to followers in a similar way to single-leader case
                        - the biggest problem are write conflicts
                        - How to avoid and resolve write conflicts
                            - One leader per session
                                - If session writes do not interfere (data stored per user for example) → avoid issue altogether

        - Converge to consistent state → apply conflict resolution policies
            - Last-write-wins policy → order writes by timestamp
            - report conflicts to the application and let it resolve it
            - Conflict free data types with specific conflict resolution logics
    - Leaderless
        - All nodes are peer in the replication network
        - any replica can read/write queries from the client
        - Writes are successful if written to W replicas and reads are successful if written to R replicas.
        - W + R > N : We expect to read up-to-date value
        - W < N : We can process writes if a node is unavailable
        - R < N : We can process reads if a node is unavailable
- Implementations of replication logs
    - Statement based
        - leader ships all write request to the follower
        - send direct INSERT or UPDATE queries
        - is non-deterministic and mostly abandoned
    - write-ahead log based
        - before actually modifying data structure, they write the intended change to an append-only write-ahead log (WAL)
        - WAL-based replication writes all changes to the leader WAL and also the followers. The followers apply the WAL entries to get consistent data.
        - Bound to the implementation of the underlying data structure. If this changes in the leader, the followers stop working
        - Postgres and Oracle use WAL-based replication.
    - logical-based
        - database generates stream of logical updates for each update to the WAL
        - These logical updates are
            - new records, the values that were inserted
            - for deleted records their unique id
            - for updated record, their id and the updated value
    - How does replica-creation work
        - take a consistent snapshot from the leader
        - shit it to the replica
        - get and id to the state of the leader's replication log at the time the snapshot was created
        - initialize the replication function to the leader id
        - the replica must retrieve and apply the replication log until it catches up with the leader.

  - Distributions of updates
    - When a write occurs in node, this write is distributed to other nodes in either of the two following nodes
    - Synchronous
      - writes need to be confirmed by a configurable number of followers before the leader reports success
        - A follower is guaranteed to have an up-to-date copy of teh data that is consistent with the leader
        - If the leader suddenly fails, we can be sure that the data is still available on the follower
        - if the synchronous follower does not respond, the write cannot be processed
        - The leader must block all writes and wait until the synchronous replica is available again
      - Impractical for all followers to be synchronous
      - up-to-date replicas but not available in case of a failure of any follower
    - Asynchronous
      - The leader reports success immediately after a write was committed to its own disk; followers apply the changes in their own pace.
        - Higher availability
          - The leader does not need to block write in case of inaccessible follower
        - A follower is not guaranteed to have an up-to-date copy of the data that is consistent with the leader
        - Writes are not guaranteed to be durable in case of leader failure
      - Complications
        - Ordering Problems
          - Clients may observe anomalous scenarios such as
            - Read-after-write → clients may not see their own write, i.e., when they connect to a replica which does not have the update
            - (non-) monotonic reads → when reading from multiple replicas concurrently a state replica might not return records that the client read from an up to date one
            - Causality violation → updates might be visible in different order at different replicas. This allows for violation of the happened before property.
      - may not have up to date data, but the system is available in the existence of failures
- CAP
  - In a replicated system, it is impossible to get all three of
    - Consistency → all nodes in the network have the same value
    - Availability → Every Request to a non-failing replica receives a response

        - Partition Tolerance → system continues to operate in the existence of component or network faults
- Consistency Models for Replicated Systems
    - Consistency models are contracts between programmers and replicated systems. i.e., it specifies the consistency between replicas and what can be observed as possible results of operations.
    - answers questions such as?
        - What are the possible results of a read query?
        - Can a client see its own updates?
        - Is the causal order of queries preserved at all copies of data?
        - Is there a total order of operations preserved at all replicas?
    - Different types of consistency models
        - Strong Consistency
            - Linearizable consistency
                - Sequential Consistency + the total order of operations conform to the real time ordering
                    - As soon as writes complete successfully the result is immediately replicated to all nodes atomically and is made available to read
                    - At any time, concurrent reads from any node return the same values.
            - Sequential Consistency
                - The operations appear to take place in some total order that is consistent with the order of operations on each replica
                - all replicas observe the same order of operations.
        - Weak Consistency
            - Client-centric consistency model
                - Provide guarantees about ordering of operations only for a single client process
                    - Monotonic reads
                        - if a process reads the value of x, any successive read operation on x by that process will always return the same value or a more recent value
                    - Monotonic writes
                        - If a process write a value to x, the replica on which a successive operation is performed reflects the effect of a previous write operation by the same process
                    - read your writes
                        - The effect of a write operation by a process on x will always be seen by a successive read operation on x by the same process.
                    - writes follow reads
                        - If a process writes a value to x following a previous read operation on x by the same process, it is guaranteed to take place on the same or

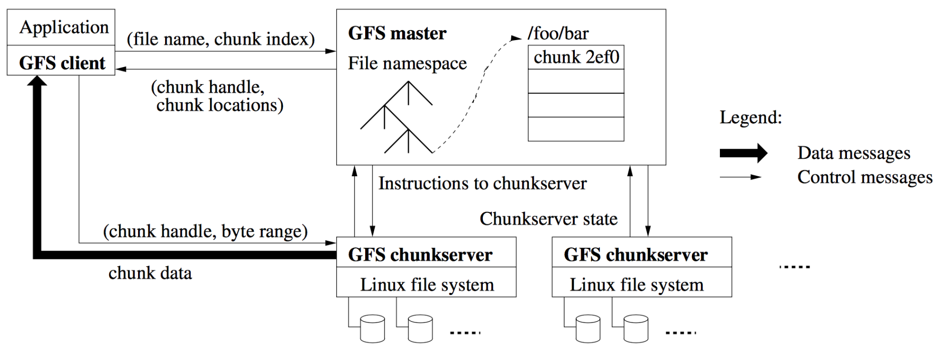
                more recent values of x that was read.
        - Causal consistency
            - causally related operations are delivered to other replicas in correct order
                - Maintain a partial order of events based on causality
            - does not require synchronous coordination across nodes and still provide sensible view of operations
        - Eventual Consistency
            - updates eventually delivered to all replicas
            - all replicas reach a consistent state if no more user updates arrive
            - Examples
                - Serach engine
                - Cloud file systems
                - Social media applications
    - Weka consistency models trade-off strong consistency for better performance
- Partitioning
    - Split the database into smaller subsets and distribute the partitions to different nodes
    - This is also known as sharding
    - Why?
        - Scalability is central
            - Queries can be run in parallel, on parts of the dataset
            - Reads and writes are spread on multiple machines
        - Partitioning is always combined with replication.
            - With partitioning, a node failure will result in irreversible data corruption
    - How?
        - Range Partitioning
            - Takes into account natural order of keys
                - split dataset in the required number of partitions based on that
            - Requires keys to be naturally ordered and keys to be equally distributed across the value range
        - Hash Partitioning
            - Calculate a hash over each item key and then produce the module of this hash to determine new partition
        - Custom Partitioning
            - Exploits locality or uniqueness properties of the data to calculate the appropriate partition to store the data to. And Example would be pre-hashed data or location specific data.
    - Request Routing
        - on partitioned datasets, clients need to be aware of the partitioning scheme in order to direct queries and writes to the appropriate nodes

        - To hide these details from the client → most partitioned systems feature a query router component sitting between the client and partitions.
        - query router knows the employed partitioning scheme and direct requests to the appropriate partitions.
    - Transactions
        - Units of work that group several reads and write to be performed together in the database.

Distributed Filesystems

- What is a filesystem?
    - system that determines how data is stored and retrieved
    - keeps track of
        - files → where the data we want to store is
        - directories → which group files together
        - metadata → length, permissions, file types, etc.
    - make sure data is always accessible and in tact
    - modern filesystems use logs
    - EXT, NTFS, APFS, ZFS
- What is a distributed file systems?
    - file system which is shared by being simultaneously mounted on multiple servers.
    - data is partitioned and replicated across the network
    - read and writes can occur on any node
    - Network file systems, such as CIFS and NFS, are not distributed file systems
        - They rely on a centralized server to maintain consistency
        - The server becomes a SPOF and a performance bottleneck
- Large-Scale File-Systems → big data revolution → need for large, distributed, high fault tolerance
    - The Google File System Paper
        - Kicked off the big data revolution
        - hardware failures are common on commodity hardware
        - files are large (G/T B) and their number is limited (millions, not billions)
        - Two main types of reads
            - large streaming reads
            - small random reads
        - workloads with sequential write that append data to files
        - once written, the files are seldom modified, they are read often sequentially
        - high sustained bandwidth trumps low latency
    - Common Distributed File Systems
        - Google File System (GFS)

- GFS Architecture

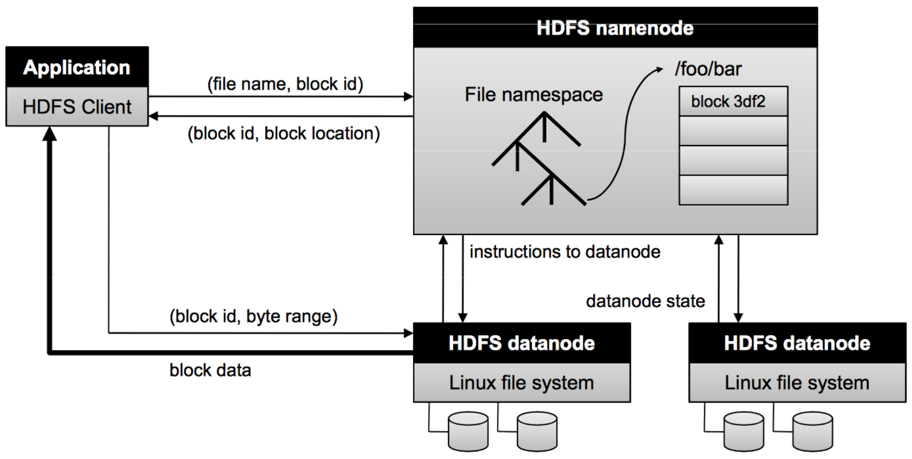


- Storage Model
  - A single file can contain many objects (e.g., Web documents)
  - Files are divided into fixed size chunks (64MB) with unique identifiers, generated at insertion time
    - Disk seek time small compared to transfer time
    - A single file can be larger than a node's disk space
    - Fixed size makes allocation computations easy
  - Files are replicated (≥ 3) across chunk servers
  - The master maintains all file system metadata (e.g., mapping between file names and chunk locations)
  - chunkservers store chunks on local disk as linux fies
    - Reading and writing of data specified by the tuple (chunk_handle, byte_range)
  - neither the client nor the chunkserver cache file data
- Reads
  - client sends file read request to GFS master
  - master replies with chunk handle and locations
  - client caches the metadta
  - client sends a data request to one of the replicas
  - the corresponding chunk server returns the requested data
- Writes
  - The client sends a request to the master server to allocate the primary replica chunkserver
  - The master sends to the client the location of the chunkserver replicas and the primary replica
  - the client sends the write data to all the replicas chunk server's buffer
  - once the replicas receive the data, the client tells the primary replica to begin the write function (primary assigns serial number to write requests)
  - The primary replica writes the data to the appropriate chunk and then the same is done on the secondary replicas
  - The secondary replicas complete the write function and reports back to the primary

        - The primary sends the confirmation to the client
    - Operations
        - Master does not keep a persistent record of chunk locations
        - queries the chunk server at start-up and then is updated by periodic polling.
        - GFS is a journaled filesystem → all operations are added to a log first then applied
            - periodically, log compaction creates checkpoints, the log is replicated across nodes
        - If a node fails
            - if it is a master → external instrumentation is required to start it somewhere else, by rerunning the operation log
            - if it is a chunkserver → it just restarts
        - chunkservers use checksums to detect data corruption
        - The master maintains a chunk version number to distinguish between up-to-date and stale replicas (which missed some mutations while its chunkserver was down)
            - Before an operation on a chunk, master ensures that version number is advanced
- Hadoop Distributed File System (HDFS)
    - Was basically meant to be an open-source implementation of GFS
    - Did change since V2
    - The are some differences
        - main one being that HDFS is a user-space filesystem written in Java

GFS	**HDFS**
Master	NameNode
Chunkserver	DataNode
operation log	journal
chunk	block
random file write	append-only
multiple writer/reader	single writer, multiple reader
chunk: 64 MB data, 32bit checksums	128MB data, separate metadata file

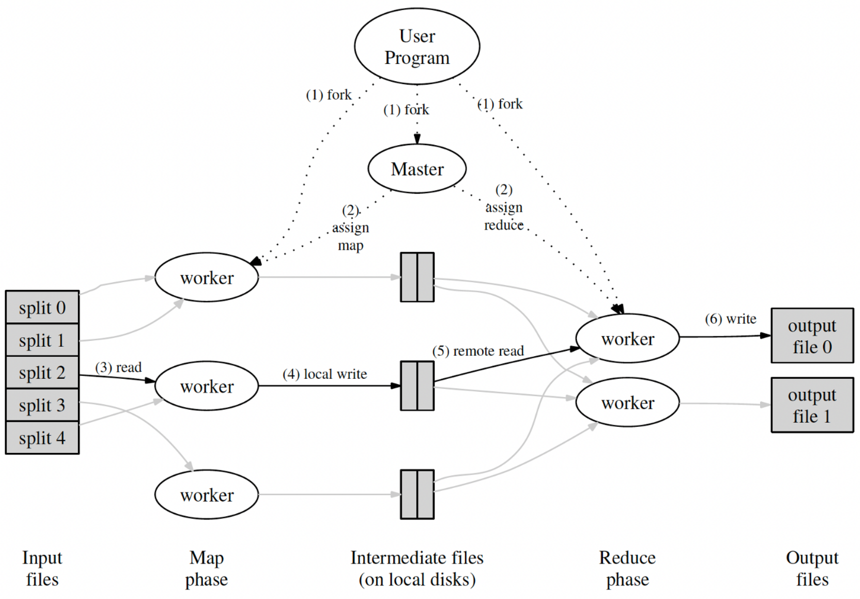
    - Architecture of HDFS



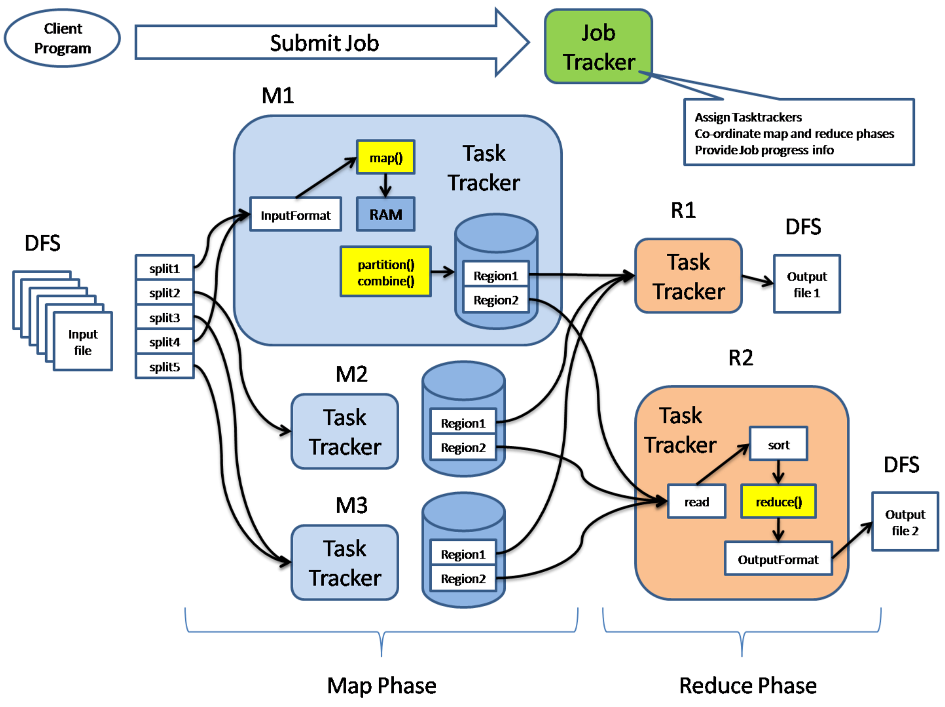
Spark: RDDs and Pair RDDS (+ Internals)

- World Before Spark
    - Parallelism
        - Parallelism is about speeding up computations by using multiple processors
        - Task Parallelism
            - Different computations performed on the same data
        - Data Parallelism
            - Apply the same computation on dataset partitions
            - Issues
                - Latency
                    - Operations are 1000x (disk) 1000000x (netowrk) slower than accessing data in memory
                - (Partial) Failures
                    - Computations on 100s of machines may fail at any time
                - i.e. → our programming model and execution should hide (but not forget!) these.
        - When processing big data, data parallelism is used to move computations close to the data instead of moving data in the network
    - Map/Reduce
        - general computation framework based on functional programming
        - assumes that data exists in a K/V store
        - the `map` and `reduce` functions supplied by the user have associated types
        - `map(K1, V1) -> List[(K2, V2)]`
        - `reduce((K2, List[V2])) -> List[(K3, V3)]`
        - Map/Reduce as a system was proposed along with GFS
        - Example Use-Cases
            - Number of occurrences of each word in a large set of documents to compute (word, N)
            - count of URL Access Frequency to compute `(targetURL, N)`
            - Reverse Web-Link Graph to compute `(targetURL, List<sourceURLs>)`
            - Filtering records
            - Top `k` records
        - Execution Overview
            - DFS chunks are assigned to Map tasks processing each chunk into a sequence of KV pairs
            - Periodically, the buffer pairs are written to local disk
            - The keys are divided among all Reduced tasks
            - Reduce tasks work on each key separately and combine all the values associated with a specific key.

- The original Implementation looks like this



- Hadoop Implements it like this



- Pros and Cons of Hadoop
  - Pros
    - Fault Tolerance
    - first framework to enable computations to run on 1000s of commodity computers
  - Cons
    - Performance

            - Before each map or reduce step, Hadoop writes to HDFS
            - Hard to express itterative problems in M/R
                - all machine learning problems are itteartive

- The World of Spark
    - What is Spark?
        - open source cluster computing framework
            - automates distribution of data and computation on a cluster of computers
            - Provides a fault-tolerant abstraction to distributed datasets
            - is based on functional programming primitives
            - provides two abstractions to data, list-like (RDDs) and table-like (Datasets)
    - Resilient Distributed Datasets (RDDs)
        - Core abstraction that Spark uses
        - make datasets distributed over a cluster of machine look like a Scala collection
        - Are immutable
        - reside (mostly) in memory
        - are transparently distributed
        - Feature all FP programming primitives
            - in addition, more to minimize shuffling
        - in practice `RDD[A]` can be seen simply as `List[A]`
        - They are Lazy
            - There are two types of operations we can do on an RDD
                - Transformation
                    - Applying a function that returns a new RDD
                    - These are lazy
                    - Common Transformations include `map` , `filter` and `flatMap`
                - Action
                    - Request the computation of a result
                    - They are eager
                    - Common Actions include `collect` , `take` , `reduce` , `fold` , and `aggregate`
                    - `collect` : Return all elements of an RDD
                        - $RDD[A].collect():Array[A]$
                    - `take` : Return the first *n* elements of the RDD
                        - $RDD[A].take(n):Array[A]$
                    - `reduce` , `fold` : Combine all elements to a single result of the same type.
                        - $RDD[A].reduce(f:(A,A) \to A):A$
                    - `aggregate` : Aggregate the elements of each partition,
                      and then the results for all the partitions

            - $RDD[A].aggr(init{:}B)(seqOp{:}(B,A)\rightarrow B, combOp{:}(B,B)\rightarrow B){:}B$
- We can also have Pair RDD
    - RDDs can represent any complex data type, if it can be iterated
    - Spark treats RDDs of type `RDD[(K,V)]` as a special named PairRDDs
    - They can be both iterated and indexed
    - Operations such as join are only defined on PairRDDs
        - we can only combine RDDs if their contents can be indexed
    - We can create Pair RDDs by applying an indexing function, using `keyBy` function, or by using records
    - functions on Pair RDDs `RDD[(K,V)]`
        - `reduceByKey`: Merge the values for each key using an associative and commutative reduce function
            - $reduceByKey(f{:}(V,V)\rightarrow V){:}RDD[(K,V)]$
        - `aggregateByKey`: Aggregate the values of each key, using given combine functions and a neutral “zero value”
            - $aggrByKey(zero{:}U)(f{:}(U,V)\rightarrow U, g{:}(U,U)\rightarrow U){:}RDD[(K,U)]$
        - `join`: Return an RDD containing all pairs of elements with matching
            - keys$join(b{:}RDD[(K,W)]){:}RDD[(K,(V,W))]$
- How does an RDD work?
    - Each RDD is characterized by five main properties
        - A list of partitions
        - A function for computing each split
        - A list of dependencies on other RDDs
        - Optionally, a Partitioner for key-value RDDs
        - Optionally, a list of preferred locations to compute each split on
    - Partitions
        - Data in RDDs is split into partitions
        - Partitions define a unit of computation and persistence
            - any spark computation transforms a partition to a new partition
        - if during computation machine fails → spark redistributes its partitions to other machines and restart the computation on those partitions only
        - Partitioning scheme of an application is configurable
            - Better configuration → better performance
        - How does this work?
            - Three partitioning schemes supported
                - Default Partitioning
                    - Splits in equally sized partitions without knowing the underlying data properties

      - Range Partitioning (Pair RDDs)
        - takes natural order of keys to split the dataset in the required number of partitions.
        - keys to be naturally ordered and keys to be equally distributed across the value range
      - Hash Partitioning (Pair RDDs)
        - Calculates a hash over each item key and then produces the module of this hash to determine the new partition.
  - Become very important when we need to run iterative algorithms
    - we could in these cases define custom partitioning schemes
    - Joins between a large, almost static dataset with a much smaller, continuously updated one
    - `reduceByKey` or `aggregateByKey` on RDDs with arithmetic keys benefit from range partitioning as the shuffling stage is minimal (or none) because reduction happens locally!

- Partition Dependencies
  - Narrow Dependencies
    - Each partition of the source RDD is used by at most one partition of the target RDD
  - Wide Dependencies
    - Multiple partitions in the target RDD depend on a single partition in the source RDD
- Shuffling
  - When operations need to calculate results using a common characteristic (e.g., common key), the data needs to reside on the same physical node.
  - The case with all “wide dependency” operations
  - Shuffling is the process of re-arranging data so that similar records end up in the same partitions
  - Shuffling is very expensive → moving data across the network and storing them to disk
- Lineage
  - RDDs contain information about how to compute themselves
    - including dependencies to other RDDs
  - RDDs basically create a DAG of computations
  - allows for RDD to be traced to its ancestors
- Persistence
  - Data Stored in three ways
    - As Java Objects
      - Each item in an RDD is an allocated object
    - As Serialized Data
      - Special (usually memory-efficient) format

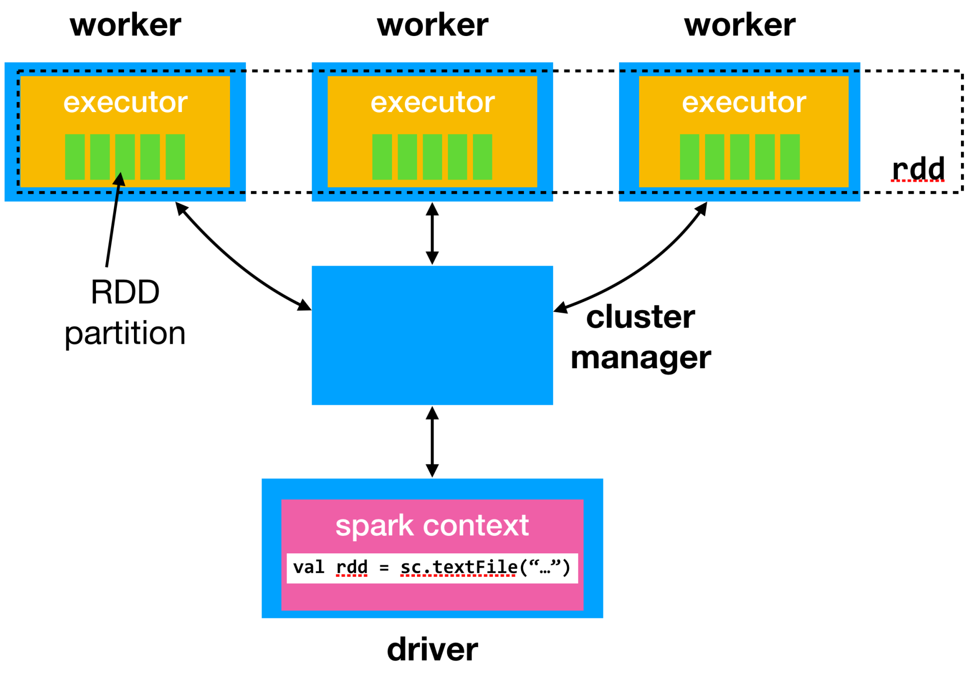
        - More CPU intensive
        - faster to send across network or to write to disk
      - On the Filesystem
        - In case of big RDDs
        - Can be mapped on a filesystem
        - usually → HDFS
    - Persistence in Spark is configurable and can be used to store frequently used computations in memory
    - e.g.

```
val rdd = sc.textFile("hdfs://host/foo.txt")
val persisted = rdd.map(x => x + 1).persist(StorageLevel.MEMORY_ONL
```

    - persisted is now cached and will not be recalculated again when used
    - Persistence storage levels

Storage level	Meaning
`MEMORY_ONLY`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
`MEMORY_AND_DISK`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
`MEMORY_ONLY_SER` \ (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). More space-efficient than deserialized objects but more CPU-intensive to read.
`MEMORY_AND_DISK_SER` (Java and Scala)	Similar to `MEMORY_ONLY_SER`, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
`DISK_ONLY`	Store the RDD partitions only on disk.

- Spark Applications
  - Spark Cluster Architecture



- Executors do the actual processing
  - worker nodes can contain multiple executors
- The driver accepts user programs and returns processing results
- The cluster manager does resource allocation
- Spark Application
  - begins by specifying the required cluster resources → the cluster manager seeks to fulfil this
  - If resources are available executors are started on worker nodes
  - Creates spark context which acts as the driver
  - Issues a number of jobs which load data partitions on the executor heap and run in threads on the executor's CPUs
  - When it finishes, the cluster manager frees the resources
- Spark Job
  - Jobs are created when an action is requested
  - Spark walks the RCC dependency graph backwards and builds a graph of stages
  - Stages are jobs with wide dependencies
    - when such an operation is requested, the spark scheduler will need to reshuffle/repartition the data
    - Stages (per RDD) are always executed serially
    - Each stage consists of one or more tasks
  - Tasks is the minimum unit of execution
    - task applies a narrow dependency function on a data partition.
    - The cluster manager starts as many job as the data partitions.
- Fault Tolerance

        - Spark uses RDD lineage information to know which partition(s) to recompute in case of node failure
        - Recomputing happens on the stage level
        - We use checkpointing to minimize recompute times
            - save job stages to reliable storage
            - checkpointing is effectively truncating RDD lineage graphs
        - Spark clusters are resilient to node failures but not master failures
        - Running spark on middle middleware such as YARN or Mesos is the only way to run multi-master setups
- RDDs and formatter data
    - RDDs by default do not impose any format on the data they store
    - if data is format, we can create a schema and have the Scala compiler type-check our computations
- RDDs can be read from various data-sources such as databases
- Some Spark Concepts
    - Broadcasts
        - are variables that allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
        - Often used to ship precomputed items
            - look-up tables
            - machine learning models
            - etc.
        - we can implement efficient in-memory joins between a processed dataset and lookup table
        - Broadcast data should be relatively big and immutable

        ```
        val curseWords = List("foo", "bar") // Use your imagination here!
        val bcw = sc.broadcast(curseWords)

        odyssey.filter(x => !curseWords.contains(x))
        ```
    - Accumulators
        - If we need to keep track of variables like performance counters, debug values, or line counts while computations are running we can use accumulators
        - This does introduce side-effects into your application and should be avoided as much as possible
        - accumulators are aggregated at the driver
        - frequent writes will cause large amounts of network traffic
        - E.g.

```
val taskTime = sc.accumulator(0L)odyssey.map{x =>  val ts = System.curr
entTimeMillis()  val r = foo(x)  taskTime += (System.currentTimeMillis
() - ts)  r
}
```

Spark SQL

- Spark Datasets
  - RDDs and structures
    - RDDs only see binary blobs with an attached type
    - Databases can do many optimization because they know the data types for each field
- Spark SQL
  - We trade some of the freedom provided by the RDD API to enable
    - declarativity, in the form of SQL
    - Automatic optimizations, similar to ones provided by databases
      - Execution plan optimizations
      - data movement/partitioning optimizations
  - We infer some cost by bringing our data to a (semi-)tabular format and describe it's schema
  - Basics
    - Spark SQL built on top of Sparks RDD and provides two main abstractions
      - Datasets, collections of strongly-types objects
      - Dataframes which are essentially a Dataset[Row]
      - SQL Syntax
    - It offers a query optimizer and an off-heap data cache
    - Can directly connect and use structured data sources and can import CSV, JSON, Parquet, Avro and other data formats by inferring their schema
  - How?
    - Similarly to normal Spark, SparkSQL needs a context object to invoke its functionality.
      - This is `SparkSession`
      - From existing `SparkContext` we can do

```
val spark = SparkSession.builder.config(sc.getConf).getOrCreate()
```

      - From RDDs containing tuples, e.g. `RDD[(String, Int, String)]`

```
import spark.implicits._
val df = rdd.toDF("name", "id", "address")
```

      - From RDDs with known complex types, e.g.

`RDD[Person]`

```
val df = persons.toDF() // Columns names/types are inferred!
```

- From RDDs, with manual schema definition

```
val schema = StructType(Array(  StructField("level", StringType, nullable = true),  StructField("date", DateType, nullable = true),  StructField("client_id", IntegerType, nullable = true),  StructField("stage", StringType, nullable = true),  StructField("msg", StringType, nullable = true),))val rowRDD = sc.textFile("ghtorrent-log.txt")             .map(_.split(" ")).             .map(r => Row(r(0), new Date(r(1)), r(2).toInt,                        r(3), r(4)))val logDF = spark.createDataFrame(rowRDD, schema)
```

- By reading (semi-)structured data files

```
val df = spark.read.json("examples/src/main/resources/people.json")
```

or

```
  val df = spark.read.format("csv").option("header", "true")    .option("inferSchema", "true").load("/airports/airlines.csv")
```

- RDDs and Datasets
  - Are Both Strongly Typed
  - Both contain objects that need to be serialized
  - the main different is that Datasets use special Encoders to convert the data internal formats that Spark can use directly when applying operations such as `map` and `filter`
    - Internal format is very efficient
    - Not uncommon to have in-memory data that use less memory spaces than their on disk format
- Operations
  - Columns
    - Can be accessed by name as `df("name")`
    - Like numpy syntactic sugar is provided such as `df("col") + 1` instead of having to write `spark.sql.expressions.Add(df("col"), lit(1).expr)`
  - Data frame Operations
    - All typical relations algebra operations are included
      - Projection

```
df.select("project_name").show(2)df.drop("project_name", "pullreq_id").show(2)
```

      - Selection

```
df.filter(df("team_size") < 4).show(2)
```

- Joins
  - DFs can be joined irrespective of the underlying implementation, as long as they share a key

    ```
    people = sqlContext.read.csv("people.csv")
    department = sqlContext.read.jdbc("jdbc:mysql://company/departmen
    ts")people.filter(people.age > 30).\
           join(department, people("deptId") === department("id")
    ```

  All types of joins are supported:

  - Left outer:

    ```
    people.join(department, people("deptId") === department("id"), "l
    eftouter")
    ```

  - Full outer:

    ```
    people.join(department, people("deptId") === department("id"), "f
    ullouter")
    ```

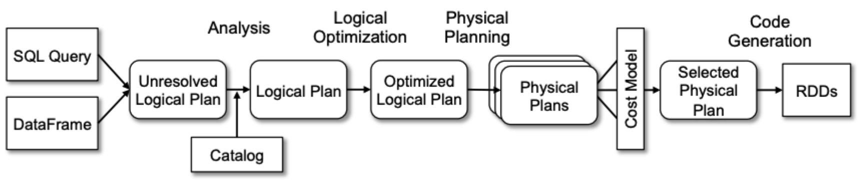
- Grouping and Aggregations
  - When `groupBy` is called on a Dataframe, is (conceptually) splint into K/V structure
    - Key is the different values of the column groupped upon and value are rows containing each individual value
  - We can only apply aggregations on groupped Dataframes (Like SQL)

    ```
    df.groupBy(df("project_name").mean("lifetime_minutes")).show()
    ```

- SQL syntax on Data Frames
  - You can register a DF as a temporary SQL view
  - You can then use SQL syntax on the data

  ```
  personsDF.createOrReplaceTempView("people")
  val res = spark.sql("SELECT * FROM people WHERE id < 10")
  res.show()
  ```

- Why Is SparkSQL so fast?
- phases of query planning in Spark SQL



        - Optimizations! (On an AST level)
        - Execution Plans
            - Analysis
                - To get to know the types of a column
            - Rule Optimization
            - Physical Optimization
                - to minimize data movement
            - Code Generation

Graph Processing

- Processing Graphs
    - graphs and hierarchical data appears every time a system models a dependency relationship
    - e.g.
        - Social graph in social networking applications
        - Web graph of linked pages
        - Dependency graph in (software) package ecosystems
- Graph Representation
    - A graph (*G*) comprises *nodes* (*V*) and *edges* (*E*). Both can carry metadata. We represent graphs as:
        - *Adjacency Matrix: An $n * n$ matrix M where a non-zero element M_{ij} indicates an edge from V_i to V_j.*
        - *Adjacency list*: A `List[(V, List[V])]` where each tuple represents a node and its connections to other nodes.
        - *Edge list*: A `List[(V, V)]` of node pairs that represents and edge
- Graph (sub-)structures
    - *Graph components*: Subgraphs in which any two vertices are connected to each other by paths.
    - *Strongly connected component*: The largest sub-graph with a path from each node to every other node
    - *Triangles* or *polygons*: A triangle occurs when one vertex is connected to two other vertices and those two vertices are also connected.
    - *Spanning trees*: A sub-graph that contains all nodes and the minimum number of edges
- Typical graph algorithms
    - Traversal
        - DFS
        - BFS
    - Node Importance
        - Calculate importance of a node relative to other nodes
        - Centrality measure or PageRank

    - Shortest Path
        - Dijkstra
        - Kruskal
        - Traveling Salesman
- Typical graph applications
    - Exploring the structure and evolution of communities or of systems of systems
    - Link prediction
        - Recommending friends
        - Recommending pages
    - Community detection
        - subgraphs with shared properties
- Approaches for graph processing
    - We can use SQL databases and recursive queries
    - Use Graph databases
    - problems with really big graphs
        - Efficiently compress the graphs that they fit in memory
        - use a message passing architecture, like the BSP model
- Application of Graphs
    - SQL-based graph traversal

```
CREATE TABLE nodes (
    id INTEGER,
    metadata ...
)
CREATE TABLE edges (
    a INTEGER,
    b INTEGER,
    metadata ...,
    CONSTRAINT src_fkey
        FOREIGN KEY (a) REFERENCES nodes(id),
    CONSTRAINT target_fkey
        FOREIGN KEY (b) REFERENCES nodes(id)
)
```

        - We model graphs as node pairs. Nodes and edges have metadata.
        - SQL-based graph traversals

```
WITH RECURSIVE transitive_closure (a, b, distance, path) AS(
    SELECT a, b, 1 as distance, a || '->' || b AS path
    FROM edges
  UNION ALL    SELECT tc.a, e.b, tc.distance + 1, tc.path || '->' || e.b
    FROM edges e
    JOIN transitive_closure tc ON e.a = tc.b
    WHERE a.metadata = ...   -- Traversal filters on nodes/edges        an
d tc.path NOT LIKE '%->' || e.b || '->%')
SELECT * FROM transitive_closure
```

- Recursive queries have a starting clause that is called on and a recursion clause

- Friend Recommendation
    - A simple social network
    - Given we are direct friends with some people, we could recommend second level friends as potential new connections
    - Recommending friends with SQL

```
WITH RECURSIVE transitive_closure (a, b, distance, path) AS(
    -- Find the yellow nodes    SELECT a, b, 1 as distance, a || '->' || b
AS path
    FROM edges
    WHERE a = src -- the blue node  UNION ALL    -- Find the red nodes
SELECT tc.a, e.b, tc.distance + 1, tc.path || '->' || e.b
    FROM edges e
    JOIN transitive_closure tc ON e.a = tc.b
    WHERE tc.path NOT LIKE '%->' || e.b || '->%'      AND tc.distance < 2
-- don't recurse into white nodes)
SELECT a, b FROM transitive_closure
GROUP BY a, b
HAVING MIN(distance) = 2 -- only report red nodes
```

    - The base expression will find all directly connected nodes, while the second will recurse into their first level descendants.

- Graph databases
    - Are specialized RDBMs for storing recursive data structure and support CRUD operations on them while maintaining transactional consistency
    - Most commonly used language for graph databases is Cypher, with the base language being Neo4J

```
(emil:Person {name:'Emil’})  <-[:KNOWS]-(jim:Person {name:'Jim’})
  -[:KNOWS]->(ian:Person {name:'Ian’})  -[:KNOWS]->(emil)
```

    - Find mutual friends of a user named Jim:

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b:Person)-[:KNOWS]->(c:Person),
(a)-[:KNOWS]->(c)
RETURN b, c
```

- Big Graphs
    - graphs → inherently recursive data structures → computations may have dependencies to previous computation steps
        - This means that they are not so trivial to parallelize
    - Traditional frameworks are not well suited for processing graphs

        - Poor locality of memory accesses
            - Access patterns not very suitable for distribution
            - Further complicated due to latency issues
        - Little work to be done per node
            - Applications mostly care about the edges
    - Case Study of PageRank
        - centrality measure based on the idea that nodes are important if multiple important nodes point to them
        - For node p_i, its page rank is recursively defined as

$$PR(p_i;0) = \frac{1}{N}$$

$$PR(p_i;t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j;t)}{L(p_j)}$$

        - where d is a damping factor (usually set 0.85) and $M(p_i)$ are the nodes pointing to p_i
        - We notice that each node updates other nodes by propagating its state
        - Simplified PageRank on Spark

```
val links: RDD[(V,List(E))] = ....cache()
var ranks = links.mapValues(v => 1.0)

for (i <- 1 to iters) {
  val contribs = links.join(ranks).values.flatMap {
    case (links, rank) =>
      val size = links.size
      links.map(url => (links, rank / size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}
```

            - The computation is *iterative* and *side-effecting* and therefore non-parallelizable. To make it side-effect free, we need to write each step of the computation to external storage.
- Bulk Synchronous Parallel Model
    - General model for parallel algorithms
    - It makes the assumption that a system has
        - Multiple processors with fast local memory
        - Pair-wise communication between processors
        - A barrier implementation (HW or other) to synchronize super steps
    - BSP supersteps
        - BSP computations are organized in supersteps
        - There are three phases

            - Local execution
                - Processors use own memory to perform computations on local data partitions
            - Data exchange / remote communication
                - Exchange of data between processes
            - Barrier Synchronization
                - Processes wait until all processes finished communicating
    - Algorithm Termination
        - Programs run until the programs stop themselves
        - Termination works as
            - Superstep 0 → all vertices are active
            - All active vertices participate in the computation at each superstep
            - A vertex deactivates itself by voting to halt
            - No execution is subsequent supersteps
            - A vertex can be reactivated by receiving a message
- Pregel
    - Is a distributed graph processing framework
    - Computations impose a BSP structure on program execution
        - computations consist of a sequence of supersteps
        - in a superstep, the framework invokes a user-defined function for each vertex
        - Function specific behaviour at a single vertex (V) and a single superstep (S)
            - It can read messages sent to V in superstep (S-1)
            - It can send messages to other vertices that will be read in superstep (S+1)
            - It can modify the state off V and its outgoing edges
    - Vertex Centric Approach
        - Algorithm iterates over vertices
            - Reminiscent of MapReduce
                - User focus on a local action
                - Each vertex is processed independently
            - By design → well suited for distributed implementation
                - All communication is from superstep S to (S+1)
                - No defined execution order within superstep
                - Free of deadlocks and data races
    - Roles
        - Graphs are stored as adjacency lists
        - Graphs are partitioned (hash) and distributed using a network filesystem
        - Two types of roles
            - Leader

        - maintains a mapping between data partitions and cluster node.
        - Implements BSP barrier
      - worker
        - For each vertex, the following stored
          - Adjacency List
          - Current Calculation Value
          - Queue of incoming messages
          - State (active/ inactive)
        - All computationally expensive operations are preformed by this role
  - Supersteps Explained
    - Workers combine incoming messages for all vertices
      - The combinator function updates the vertex state
    - If a termination condition has been met, the vertex votes to exclude itself for further iterations
    - The vertex optionally updates a global aggregator
    - Messages are passed
      - if receiving vertex is local → its message queue is updated
      - else → wraps messages per receiving node and send them in bulk
- Spark GraphX
  - is a new component of Spark for graphs and graph-parallel computation
  - It provides a variant of Pregel’s API for developing vertex-centric algorithms
  - Spark uses it’s underlying fault tolerance, checkpointing, partitioning and communication mechanism to store the graph
    - Fault tolerance
      - The fault tolerance model is reminiscent of spark
      - Periodically, the leader instruct the workers to save the state of their in-memory data to persistent storage
      - worker failure detected through keep-alive messages the leader issues to workers
      - In case of failure, the leader reassigns graph partitions to live workers
        - They reload their partition state from the most recently available checkpoint.
  - GraphX Allows for operation on the underlying data structures as both a graph using graph concepts and processing primitives as well as separate collections of edges and vertices that can be transformed using FP primitives.

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
  . . .
}
```

  - Pregel API in Spark

```
def pregel[A](
    // Initialization message
    initialMsg: A,
    // Max super steps
    maxIter: Int = Int.MaxValue,
    activeDir: EdgeDirection = EdgeDirection.Out,
    // Program to update the vertex
    vprog: (VertexId, VD, A) => VD,
    // Program to determine edges to send a message to
    sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
    // Program to combine incoming messages
    mergeMsg: (A, A) => A
) : Graph[V, E]
```

- PageRank with Pregel/Spark

```
val pagerankGraph: Graph[Double, Double] = graph
  .mapVertices((id, attr) => 1.0) // Initial Pagerank for nodes

def vertexProgram(id: VertexId, attr: Double, msgSum: Double): Double =
  resetProb + (1.0 - resetProb) * msgSum
def sendMessage(id: VertexId, edge: EdgeTriplet[Double, Double]): Iterator[(Ver
  Iterator((edge.dstId, edge.srcAttr * edge.attr))
def messageCombiner(a: Double, b: Double): Double = a + b
val initialMessage = 0.0

// Execute Pregel for a fixed number of iterations.
Pregel(pagerankGraph, initialMessage, numIter)(
  vertexProgram, sendMessage, messageCombiner)
```

Stream Processing

- What is streaming?
  - Typical big data scenario
    - Aggregate data in intermedia storage
    - Running batch job overnight → store results in permanent storage
    - User spark for interactive exploration of recent data
  - Data is not static
    - running processes are continuously generating data
    - Users need to continuously monitor processes
  - We distinguish between two types of data
    - Bounded
      - A dataset that can be enumerated and/or iterated upon
    - Unbounded
      - A dataset that we can only enumerate given a snapshot.

            - does not have a size property
    - We also distinguish between two types of processing
        - Batch Processing
            - Apply algorithm on a bounded dataset to produce a single result at the end
            - Unix, Map/Reduce and Spark
        - Stream Processing
            - Apply algorithm on continuously updating data and continuously creates results
            - Flink and Storm are stream processing system
            - Natural fit for unbounded data
            - Bounded data usually a time-restricted view of unbounded data
            - use-cases
                - Intrusion and fraud detection
                - Algorithmic trading
                - Process Monitoring
                    - production processes
                    - logs
                - Traffic monitoring
                - Discarding raw data and prefer to store aggregates
            - What can we learn from Unix
                - What it has
                    - Streaming data acquisition → tail or pipe
                    - Intermedia storage → pipes
                    - Ways of applying function on streaming data
                - What it still needs
                    - Splitting streams in batches (windowing)
                    - Recomputing when new batches arrive (triggers)
- Stream Processing 101
    - SP → set of techniques and corresponding systems that process timestamped events
    - For it to work we need to major things
        - A component that acquires events from producers and forwards it to consumers
        - A component that processes events
    - Both components need to be scalable, distributable, and fault-tolerant
    - Messaging Systems
        - SP fundamentally deal with events
        - Events are produced by continuous processes and to be processed they must be consumed
        - Messaging Systems solve the problem of connecting producers to consumers.
        - We Identify several types of messaging systems

- Publish / Subscribe Systems
    - connect multiple producers to multiple consumers
    - Direct messaging systems
        - simple network communication → broadcast messages to multiple consumers
        - Fast but require the producers / consumers to deal with data loss
    - Message brokers / Queues
        - centralized systems that sit between producers and consumers and deal with the complexities of reliable message delivery.
- Broker-based messaging
    - We Distinguish two roles
        - The broker:
            - Buffers the messages, spilling to disk as necessary
            - Routes the messages to the appropriate queues
            - Notifies consumers when messages have arrived
        - The consumers:
            - Subscribe to a queue that contains the desired messages
            - Ack the message receipt (or successful processing)
    - Sends messages in any of the following modes
        - Fire and Forget
            - The broker acks the message immediately
        - Transaction-based
            - The broker writes the message to permanent storage prior to ack-ing it
    - Messaging patterns
        - We define a competing worker as multiple consumers reading from a single queue → i.e., competing for incoming resources
        - Fan Out Pattern
            - Each consumer has a queue of its own
            - Incoming messages are replicated on all queues
        - Topics Patterns
            - The producer assigns keys to message metadata
            - The consumer creates topic queries by specifying the keys it is interest to receive messages for
    - Drawbacks
        - We know that after a message is received, it disappears
        - Leads to Lost opportunities
            - No reprocessing of messages
            - No proof of messages having been delivered
- Log-based messaging

        - Logs → append-only data structure stored on disk
        - We can use logs to implement messaging systems
        - How?
          - Producers append messages to the log
          - All consumers connect to the log and pull messages from it
            - A new client starts processing from the beginning of the log
          - To maximize performance → the broker partitions and distributes the log to a cluster of machines
          - The broker keeps track of the current messages offset for each consumer per partition.
- Programming Models for Stream Processing
  - We require models for processing streams that enable processing of events to derive state
  - We Distinguish three types of models
    - Event sourcing / Common Query Segregation (CQS)
      - Essentially → capture all changes to an application state as a sequence of events.
      - Store the event that caused the mutation in an immutable log instead of mutating the application state.
      - Application state is generated by processing the events.
      - With this we can
        - use specialized systems for scaling writes and reads while the application remains stateless
        - Provide separate and continuously updated views of the application state based on selection criteria
        - Regenerate the application state at any point in time be reprocessing events
    - Reactive Programming
      - declarative programming paradigm dealing with data streams and propagation of change
      - We look at the event sources as infinite collections on which observers subscribe to receive events
    - The DataFlow Model
      - Attempts to explain stream processing in four dimensions
        - What: results are being computed
          - We distinguish two types of events on streams
            - Element-wise → ops apply a function to each individual message
              - `map`, `filter` , `merge` , `flatMap` , `keyBy` , and `join`
            - Aggregation → group multiple events together and apply reduction

```
Stream[A].aggregate(f: AggregationFunction[A, T, B]): Stream[

trait AggregationFunction[IN, ACC, OUT] {
  def createAccumulator(): ACC
  def add(value: IN, acc: ACC): ACC // Type conversion IN -> 
  def getResult(acc: ACC): OUT      // Type conversion ACC ->
  def merge(a: ACC, b: ACC): ACC
}
```

- Where: in event time they are being computed
  - Windows are static size or time-length "batches" of data
  - Session Windows
    - dynamically sized windows that aggregate batches of group activities. Windows end after session gap time
  - There are 2 things to remember when using event-time windows.
    - **Buffering:** Aggregation functions are applied when the window finishes. This means that in-flight events need to be buffered in RAM and spilled to disk.
    - **Completeness:** Given that events may arrive out of order, how can we know that a window is ready to be materialized and what do we do with out of order events?
- When: in processing time they are materialized
  - A trigger defines when in processing time the results of windows are materialized / processed.
  - We define these types of triggers
    - Per-record triggers
      - fire after x records in a window have been encountered
    - Aligned delay trigger
      - fire after a specified amount of time has passed across all active windows
    - unaligned delay trigger
      - fire after a specified amount of time has passed after the first event in a single window
  - Watermarks?
    - Event-time processors need to determine when event time has progressed enough so that they can trigger windows
    - When reprocessing events from storage a system might process weeks of event-time data in seconds
    - relying on processing time to trigger windows is not enough
    - Watermarks flow as part of the data stream and carry a timestamp
      - Declaration that by a point in the stream, all events carrying a timestamp up to the watermark should have arrived
      - allow late messages to be processed up to a specified amount delay
    - As the watermarks flow through the streaming program they advance the event time at the operators where they arrive
      - Whenever an operator advances its event time it generates a new watermark downstream for its successor operators

    - How: earlier results relate to later refinements
        - In complex cases
            - combination of triggers and watermarks flowing may cause a window to be materialized multiple times
            - In such cases we can discard, accumulate or accumulate and retract the window results
- The notion of time in stream processing
    - There are two notions of time
        - Processing time → the time at which events are observed in the system
        - Event time → the time at which events occurred
    - Application calculating streaming aggregates don't care much about the event order
    - Applications with precise timing requirements care about event time
        - Keep in mind that events can still enter the system with a delay or out of order
    - Event Time Skew
        - if the processing time is t
            - skew is calculated t-s where s is the timestamp of the latest event processed
            - lag is calculated as t -s where is the actual timestamp of an event