HLD S3 + Quad trees (nearest neighbors)

Contents

[Problem Statement: 2](#_Toc184747075)

[File Upload/Download Systems and Graceful Failure Handling 2](#_Toc184747076)

[Designing a file upload/download system 4](#_Toc184747077)

[Distributed file systems 7](#_Toc184747078)

[Distributed File Systems and SDFS Concepts 8](#_Toc184747079)

[Download Flow Explanation 11](#_Toc184747080)

[Nearest Neighbour 14](#_Toc184747081)

[Nearest Neighbour Problem 16](#_Toc184747082)

## Problem Statement:

You need to design a website that supports **upload and download** of **very large files** (e.g., 2 TB, 50 TB, 10 TB). The system should address the following challenges:

**Key Challenges:**

1. **Storage:**
   * **Scalability:** Files may not fit on a single machine. Need a way to store and manage such large files across multiple machines (e.g., distributed storage systems).
   * **Efficient Space Usage:** Prevent dedicating entire storage to a single file, ensuring system scalability for multiple users.
2. **Reliability and Durability:**
   * Files should never be lost due to hardware failures or other issues.
   * Need mechanisms like **replication** and **backup** to ensure data durability.
3. **Handling Network Failures (Graceful Retry):**
   * Large file uploads/downloads are prone to network disruptions.
   * Implement **partial uploads/downloads** so progress isn't lost in case of failure.
   * Ensure **resume capability** for interrupted file transfers (e.g., similar to torrent systems).
4. **File Accessibility:**
   * Uploaded files must be retrievable/downloadable by the user reliably and efficiently.

## File Upload/Download Systems and Graceful Failure Handling

**Key Concepts**

1. **Graceful Handling of Failures**:
   * Ensure progress is preserved during failures.
   * For example, if downloading a large file (e.g., 1 GB) and a failure occurs after downloading 250 MB, the process should resume from 250 MB rather than restarting from 0.
2. **Chunking Large Files**:
   * Large files are divided into smaller manageable chunks.
   * Each chunk is uploaded or downloaded independently.
   * Only the failed chunks are retried, saving bandwidth and time.
3. **Parallelism in File Handling**:
   * File chunks can be uploaded/downloaded in parallel.
   * This approach accelerates the process and makes better use of network resources.

**Real-World Implementations**

1. **Google File System (GFS) and Hadoop Distributed File System (HDFS)**:
   * Distributed file systems for storing and handling large files across multiple machines.
   * Used for reliable and scalable storage solutions.
2. **Data Processing with Large Files**:
   * Tools like **Apache Spark** and **MapReduce** are used for processing large datasets, such as log files or analytics tasks.
   * These frameworks enable distributed computation over large files to derive insights efficiently.

**Examples of File Download Systems**

1. **Torrent Protocol**:
   * Uses a distributed approach to file sharing.
   * **Seed Users**: Users who already have the file being shared.
   * **Metadata in Torrent Files**:
     + Details of file chunks.
     + Identifiers for locating seed users with the desired file.
   * **Chunking in Torrents**:
     + Files are divided into multiple chunks (e.g., 800 MB file into 800 1-MB chunks).
     + Chunks are downloaded from multiple seed users simultaneously, improving speed.
   * **Fault Tolerance**:
     + If a seed user is slow or unavailable, chunks are reassigned to faster users to maintain overall speed.
     + Download resumes for failed or incomplete chunks rather than restarting the entire process.

**Advantages of Chunk-Based Systems**

* **Efficiency**:
  + Chunks allow independent handling of smaller parts, optimizing bandwidth and resource utilization.
* **Fault Tolerance**:
  + Partial downloads can resume from where they stopped, ensuring no progress is lost during failures.
* **Parallel Downloads**:
  + Chunks can be downloaded from multiple sources simultaneously, maximizing network speed.

**Lessons from Torrents:**

* Breaking files into chunks enhances resilience and speed.
* Parallel downloading improves performance when dealing with multiple sources.
* Metadata-driven systems enable efficient tracking and management of file chunks.

## Designing a file upload/download system

**1. Why Chunk Files?**

* **Large File Handling:** Uploading/downloading a large file in one go can lead to failures and inefficiencies due to network interruptions or memory constraints.
* **Parallelism:** Breaking files into chunks allows simultaneous uploads/downloads, enhancing speed.
* **Retry Mechanisms:** If a chunk fails, only the failed chunk needs to be retried.

**2. Ideal Chunk Size Considerations**

The choice of chunk size involves trade-offs. Here's what to consider:

**Small Chunk Sizes**

* **Pros:**
  + Easier retries: Less data needs to be re-sent on failure.
  + Better granularity for parallel uploads.
* **Cons:**
  + High metadata overhead: Each chunk requires additional metadata (e.g., size, checksum).
  + Increased database/storage mapping complexity.
  + Network inefficiency: Small chunks can lead to higher protocol overhead.

**Large Chunk Sizes**

* **Pros:**
  + Reduced metadata and mapping overhead.
  + Fewer network requests, improving efficiency.
* **Cons:**
  + Expensive retries: A failure involves retransmitting a larger amount of data.
  + Higher memory requirements for processing.

**Balanced Chunk Size**

The ideal chunk size minimizes both metadata overhead and retry costs while aligning with system constraints.

**3. Factors Affecting Chunk Size**

* **Default HTTP Limits:**
  + Many systems have a default HTTP request body size limit (e.g., 4 MB). Exceeding this requires explicit configurations.
* **Operating System Block Sizes:**
  + Filesystems store data in fixed-size blocks (e.g., 64 MB or 128 MB). Aligning chunk size with block size reduces fragmentation and improves performance.
  + Example: HDFS default block sizes (64 MB in v1.0, 128 MB in v2.0).
* **Use Case and Network Conditions:**
  + High-latency or unstable networks may favor smaller chunks for reliability.
  + Scenarios requiring faster transfers and fewer retries (e.g., streaming) may use larger chunks.
* **Hardware Capabilities:**
  + RAM and disk I/O speed can dictate the maximum practical chunk size.
* **Concurrency Goals:**
  + Smaller chunks can better utilize parallelism in systems with high concurrency needs.

**4. Dynamic vs. Static Chunk Sizes**

* **Dynamic Chunk Sizes:**
  + Adapt to file size: Small files can use smaller chunks; large files use larger chunks.
  + Allows better optimization but adds complexity to implementation.
* **Static Chunk Sizes:**
  + Simpler to implement and maintain.
  + Ensures uniform behavior across files.

**5. Metadata Considerations**

Every chunk requires associated metadata:

* **Size:** Helps in reassembling the file.
* **Checksum:** Ensures data integrity.
* **Location Mapping:** Links chunks to storage servers or blocks.

Minimizing metadata overhead is critical, especially when dealing with small chunks.

**6. Hashing for Integrity (Checksums)**

* Use standard algorithms like **MD5**, **SHA-256**, etc., to generate checksums for data integrity.
* Checksum verification ensures correctness after transfer.
* Hashing can be performed in **O(N)** time alongside the data transfer, making it efficient.

**7. Recommendations**

* Default to a chunk size aligned with OS block sizes (e.g., 64 MB or 128 MB) for large files.
* Allow configuration for dynamic adjustment based on file size.
* Consider network and storage constraints while tuning chunk sizes.
* Ensure efficient metadata handling to avoid excessive overhead.

## Distributed file systems

**1. Challenges with File Storage in Distributed Systems**

* **File Size Limitations**: A single machine cannot store very large files (e.g., 50TB) due to storage constraints.
* **Fault Tolerance**: Storing data on a single machine creates a single point of failure.
* **Scalability**: As the number of files increases (e.g., in platforms like Instagram or test data management systems), a single machine will run out of storage.

**2. Breaking Files into Chunks**

* Files are divided into smaller, fixed-size chunks (e.g., 64MB each).
* Chunks are distributed across multiple machines for scalability and redundancy.

**3. Replication for Fault Tolerance**

* Each chunk is replicated and stored on multiple machines (e.g., three replicas per chunk).
* This ensures data availability even if one or two machines fail.

**4. Metadata Management**

* A **centralized metadata store** (e.g., a "NameNode" in HDFS) keeps track of:
  + Which chunks belong to which files.
  + The machines where each chunk is stored.
* This metadata allows the system to locate and retrieve file chunks during downloads or reassemble the file from its parts.

**5. Replication and Optimization**

* When storing chunks, the system considers:
  + Machine storage capacity.
  + Load on machines.
* The NameNode (or equivalent central store) may use algorithms like consistent hashing or load balancing to distribute chunks effectively.

**6. Handling Uploads**

* Chunks can arrive at the server in any order.
* Metadata (file and chunk mapping) is first sent to the NameNode.
* The NameNode determines the machines where each chunk should be stored and communicates this to the client.

**7. Minimizing Single Points of Failure**

* Replicas of the NameNode are maintained to avoid dependency on a single machine.
* If the NameNode fails, its replica ensures metadata availability.

**8. Potential for Eliminating the NameNode**

* Theoretically, it’s possible to eliminate the NameNode by:
  + Using consistent hashing to determine chunk placement dynamically.
  + Storing metadata (file-to-chunk mappings) in a distributed, decentralized manner.
* However, this can introduce additional complexity in managing metadata and consistency.

## Distributed File Systems and SDFS Concepts

**Overview of SDFS (Simplified Distributed File System):**

1. **Components:**
   * **NameNode:**
     + A central orchestrator responsible for managing the metadata, replication, and mapping of chunks to machines.
   * **DataNodes:**
     + Machines that store chunks of data. They act as storage units and do not perform orchestration.
   * **Chunks:**
     + Large files are broken into smaller pieces called chunks, typically done by the client.
2. **Core Features:**
   * **Replication:**
     + SDFS maintains multiple copies (e.g., 3 copies) of each chunk for fault tolerance.
     + Replication is managed by the NameNode.
   * **Under-replication:**
     + Occurs when the number of replicas for a chunk drops below the configured level (e.g., machine failure).
     + NameNode identifies under-replicated chunks and reallocates replication tasks to maintain the required number of replicas.

**Terminologies:**

1. **Rack Awareness:**
   * Ensures that replicas are distributed across different racks.
   * Prevents all replicas of a chunk from being lost if a single rack fails.
   * Racks are vertical stacks of machines connected via a shared router; racks share connectivity through a switch.
2. **Disaster Recovery (DR):**
   * Allows replication across geographically separated data centers for resilience.
   * Controlled by the NameNode based on configuration flags.
3. **Zookeeper:**
   * Used to track the status of DataNodes.
   * Maintains **ephemeral nodes** for DataNode health checks.
     + If a machine fails, its ephemeral node disappears, notifying the NameNode of the failure.

**Workflow Examples:**

1. **Handling Under-replication:**
   * **Scenario:**
     + Machine 5 goes offline, which stored chunks C2 and C3.
     + The replication level for these chunks drops below the required count (e.g., 2 copies instead of 3).
   * **Resolution:**
     + The NameNode:
       - Checks its metadata to identify chunks affected by the failure.
       - Assigns new DataNodes (e.g., Machine 4 and Machine 3) to host the under-replicated chunks.
       - Initiates the replication process to restore the required level.
2. **Chunk Allocation:**
   * The NameNode assigns chunks to specific DataNodes based on:
     + Rack awareness.
     + Load balancing to distribute chunks evenly across machines.
3. **Replication Update:**
   * If a chunk replica is deleted or corrupted, the NameNode ensures a new replica is created on another DataNode.

**Advantages of Centralized Orchestration:**

1. **Rack Awareness:**
   * Ensures data redundancy across physical infrastructure.
2. **Disaster Recovery:**
   * Allows replicas to exist outside the primary data center.
3. **Efficient Recovery:**
   * Centralized metadata enables rapid detection and correction of under-replication.

**Challenges of Centralized Systems:**

1. **Single Point of Failure:**
   * The NameNode's availability is critical to the system’s operation.
2. **Scalability Issues:**
   * Centralized control can become a bottleneck as the number of files and chunks increases.

**Client-side Considerations:**

1. **File Chunking:**
   * Clients break large files into smaller chunks during upload (e.g., multipart upload in S3 or browsers).
   * Libraries and standards like multipart/form-data or MIME are used to manage chunking.
2. **Encoding and Upload Mechanisms:**
   * Uploads are typically facilitated using standards (e.g., HTML enctype="multipart/form-data").

**Summary of Responsibilities:**

* **NameNode:**
  + Maintains mappings of chunks to machines.
  + Monitors DataNodes (via Zookeeper).
  + Handles replication and recovery tasks.
* **DataNodes:**
  + Store and manage chunks.
  + Report health and status to the NameNode.

## Download Flow Explanation

**Overview**

* **Use Case**: Downloading a file (e.g., an image or video) from a website, without considering CDNs (Content Delivery Networks).
* **Key Concepts**: Metadata, chunked file downloads, and retry mechanisms.

**Steps in the Download Flow**

1. **Client Request**:
   * A client initiates a request to download a file (e.g., Mission Impossible video or an image).
   * The server responds with **metadata** about the file:
     + Number of chunks.
     + Size of each chunk.
     + Locations of the chunks.
2. **Fetching Metadata**:
   * The **App Server** retrieves the metadata of the requested file from a **NameNode Server**.
     + NameNode provides:
       - File chunks and their identifiers.
       - The physical or virtual machines where the chunks are stored.
3. **Chunk Download**:
   * The client starts downloading chunks in parallel from various storage nodes.
   * This allows efficient handling of large files.

**Retry and Resumption Mechanism**

* **Chunk-based Retry**:
  + If a network failure occurs:
    - Retry is done at the chunk level, not for the entire file.
    - Partially downloaded chunks can also be resumed.
  + Example: Chrome browser supports resuming downloads.
    - Already downloaded chunks are preserved.
    - Only the remaining chunks are retried.
* **Graceful Handling**:
  + The server and client communicate using chunk identifiers for partial downloads.
  + This reduces redundant data transfer and saves bandwidth.

**Key Features of Chunked Downloads**

1. **Metadata-Driven**:
   * Metadata helps organize chunks and keep track of retries.
2. **Parallel Processing**:
   * Chunks can be downloaded simultaneously from multiple storage nodes.
   * Improves download speed and efficiency.
3. **Fault Tolerance**:
   * Chunk-based retries minimize failure impact.
   * Partial resumption is supported, ensuring robust downloads.

**Philosophical Perspective**

* **Large File Handling**:
  + Traditional **request-response models** are insufficient for large file downloads due to:
    - Increased probability of failures.
    - Long request durations.
* **Solutions**:
  + **Break into chunks**: Enable multi-part requests to manage large files.
  + **Graceful retries**: Handle failures by retrying specific chunks.

**Interview Context**

* **Common Question**:
  + How to handle long-running downloads or uploads in a way that minimizes failure and supports retries?
* **Key Answer**:
  + Implement chunking.
  + Use metadata to manage retries and resumption effectively.

**Takeaways**

* **Chunked Downloads**:
  + Ensure efficient, scalable, and fault-tolerant file transfers.
  + Align well with modern web and application practices.
* **Role of Metadata**:
  + Central to managing chunks and their locations.
* **Real-World Examples**:
  + Chrome browser, torrent protocols, and HDFS (Hadoop Distributed File System) follow similar principles.

## Nearest Neighbour

You need to find the nearest X restaurants (or other entities) from a given latitude and longitude. The brute-force approach of calculating the distance for all restaurants is inefficient for large datasets, so optimizations are required.

**Brute-Force Approach**

* Iterate over all restaurants.
* Calculate the distance for each using the Haversine formula (for geospatial data) or Euclidean distance (for 2D maps).
* Use a max-heap of size X to store the nearest restaurants.
* Time complexity: , where N is the total number of restaurants.

This is accurate but computationally expensive for large datasets.

**Optimization Strategies**

**1. Geospatial Indexing (Spatial Databases)**

* Use a spatial database like **PostGIS** or **SpatiaLite**, which adds geospatial capabilities to SQL databases.
* These systems support indexing methods like **R-trees** or **Quad-trees**, designed for spatial queries.
* Example SQL query:

|  |
| --- |
| SELECT \*  FROM restaurants  ORDER BY ST\_Distance(ST\_Point(lon, lat), ST\_Point(current\_lon, current\_lat))  LIMIT X; |

**2. K-d Trees**

* A **K-dimensional tree** is a binary space-partitioning data structure.
* It recursively divides space into k-dimensional rectangles, allowing efficient nearest-neighbour queries.
* Complexity: for a well-balanced tree.

**3. Grid Partitioning**

* Divide the geographic space into a grid of cells.
* Store restaurants in the respective grid cells.
* For a query, check the cell containing the point and neighbouring cells to find the nearest neighbours.

**4. Caching and Precomputing**

* For static datasets or infrequent updates, precompute distances for common queries.
* Cache results to serve repeated queries efficiently.

**5. Hierarchical Queries**

* If precise distances aren't immediately necessary, use hierarchical queries:
  + First, filter restaurants by bounding box (a rough rectangular area).
  + Then refine the search using exact distance calculations.

**Composite Indexing in Databases**

* Create composite indexes on latitude and longitude:

CREATE INDEX idx\_lat\_lon ON restaurants (latitude, longitude);

* The database will internally sort entries and create a B+-tree or similar structure.
* This allows efficient filtering of ranges (e.g., latitude BETWEEN ... AND ...).

**Challenges**

* Composite indexes work well for sequential scans but aren't optimal for geospatial distance-based queries. Combining this with a spatial index yields better performance.

## Nearest Neighbour Problem

**Problem Statement**

Given a latitude and longitude, the task is to find the nearest restaurants efficiently.

**Approaches Discussed**

**1. Brute Force**

* Check every restaurant’s distance from the given latitude and longitude.
* **Issues:**
  + Computationally expensive.
  + Extremely slow for large datasets.

**2. SQL Query**

* Use a SQL query with conditions like:
  + WHERE X BETWEEN ... AND ...
  + OR Y BETWEEN ... AND ...
* **Optimization:**
  + Composite Index can optimize either the X or Y query, not both.
* **Drawbacks:**
  + Still suboptimal as the second coordinate requires scanning all values.
  + Slight improvement over brute force but not efficient.

**3. Grid-Based Approach**

* **Concept:**
  + Divide the world into equal-sized grids.
  + Assign each restaurant a gridID based on its latitude and longitude.
  + Query using:

SELECT \* FROM restaurants WHERE gridID IN (neighboring\_gridIDs);

* + Use an **index on gridID** for efficient querying.
* **Steps:**

1. **Determine Grid ID:**
   * + Divide the latitude and longitude into steps to calculate the gridID.
     + Example:
       - If grids are sized X, compute grid using:

|  |
| --- |
| row\_number = (latitude + 180) / grid\_height  column\_number = (longitude + 180) / grid\_width |

1. **Find Neighbouring Grids:**

* For a grid at ID g:
  + - Left: g-1
    - Right: g+1
    - Above: g-1,000,000 (assuming 1,000,000 grids per row)
    - Below: g+1,000,000

1. **Query** neighbouring grids to retrieve results.

* **Advantages:**
  + Faster than brute force and SQL approaches.
  + Efficient for dense regions (e.g., urban areas).
* **Limitations:**
  + Uniform grid sizes lead to inefficiencies:
    - Sparse grids (e.g., oceans) have no restaurants.
    - Dense grids (e.g., cities) may still require expanding the search area.

**4. Dynamic Grid Sizes**

* **Problem:**
  + Equal grid sizes are inefficient due to varying restaurant densities.
* **Proposed Solution:**
  + Use variable-sized grids:
    - Smaller grids in high-density areas.
    - Larger grids in low-density areas.
  + **Challenges:**
    - Determine how to:
      * Locate a grid for a given point.
      * Identify neighbouring grids efficiently.

**5. Recursive Subdivision: Quadtree**

* **Concept:**
  + Start with a single grid for the entire world.
  + Recursively subdivide into 4 smaller grids if a grid has more than a threshold number of restaurants (e.g., 50).
  + Stop subdividing when all grids contain fewer than the threshold.
  + **Result:**
    - A tree structure where each node has 4 children (quadtree).
* **Advantages:**
  + Handles both dense and sparse regions efficiently.
  + Adapts to restaurant density dynamically.
* **Extension to Higher Dimensions:**
  + Use a **KD-Tree** (K-dimensional tree) for multi-dimensional data.