RAID Multimedia Storage Performance Simulator

Complete Technical Documentation

Team Information

Project Team Members

Name	Student ID
Saif Elden Ahmed Lotfy	23030241
Khaled Atef Mahmoud	23030064
Mohamed Waleed Abd Elgwad	23030176
Omar Ahmed Omar	23030122
Saeed Mamdouh Mohamed	23030086
Mayada Salah	23030198
Youssef Ahmed Hassan	23030218

Executive Summary

The RAID Multimedia Storage Performance Simulator is a comprehensive Python-based application designed to analyze and simulate Redundant Array of Independent Disks (RAID) storage configurations. This project demonstrates the practical implementation of disk performance formulas, RAID calculation methodologies, and storage system design principles taught in the Information Storage Systems course.

The application provides an interactive web-based interface using Gradio, allowing users to:

- Scan and analyze multimedia file collections
- Configure various RAID levels (RAID 0, RAID 1, RAID 5)
- Calculate disk performance metrics using formulas from course materials
- Simulate read/write operations and generate performance reports
- Visualize capacity distribution and performance metrics
- Export results as CSV reports and virtual disk structures

This documentation covers the architecture, implementation details, technical formulas, and usage instructions for the complete system.

Table of Contents

1. Project Overview
2. System Architecture
3. Technical Formulas
4. Module Documentation
5. Installation and Setup
6. Usage Guide
7. Code Snippets
8. Performance Metrics
9. Results and Visualization
10. Troubleshooting

Project Overview

Purpose

This project serves as an educational tool to understand:

- How RAID systems distribute data across multiple disks
- Disk performance calculations and their impact on system design
- Capacity planning and redundancy trade-offs
- The relationship between storage efficiency and fault tolerance

Key Features

- **Interactive Web UI**: Built with Gradio for ease of use
- **Multi-RAID Support**: Supports RAID 0, RAID 1, and RAID 5 configurations
- **Disk Performance Analysis**: Implements formulas from course materials (sec-2-1.pdf)
- **RAID Calculations**: Implements formulas from course materials (sec-5-1.pdf)
- **Virtual Disk Simulation**: Creates realistic file distributions across disk arrays
- **Comprehensive Reporting**: Generates CSV reports, charts, and statistics
- **Export Capabilities**: Download results in multiple formats

Technology Stack

- **Language**: Python 3.7+
- **Web Framework**: Gradio 3.x
- **Data Analysis**: Pandas
- **Visualization**: Matplotlib, PIL
- **File Handling**: pathlib, os, shutil

System Architecture

Project Structure

```
Information-Storage-Project/
├── run.py # Main application entry point (1,200+ lines)
├── Modules/
│ ├── Disk_Performance.py # Disk performance calculations
│ ├── Raid_Calculation.py # RAID capacity and efficiency formulas
│ ├── Raid_Simulation.py # Performance simulation logic
│ └── Data_Analysis.py # Statistical analysis and reporting
├── FORMULAS.md # Complete formula documentation
├── README.md # Project overview
├── docs/ # Additional documentation
└── reports/ # Generated output files
```

Application Flow

The application follows a three-stage workflow:

Stage 1: Load

- User selects a folder containing multimedia files
- Application scans folder recursively for supported file types
- Files are analyzed for metadata (size, resolution, type)
- Results are displayed in an interactive table

Stage 2: Config

- User selects RAID level (RAID 0, RAID 1, RAID 5)
- User selects number of disks (2-12 disks)
- Application calculates and displays theoretical metrics
- Preview of usable capacity and efficiency percentages

Stage 3: Run

- Complete simulation executes with selected configuration
- Disk performance metrics are calculated using formulas
- Performance simulations generate read/write times
- Virtual disk structures are created
- Reports and visualizations are generated
- Files are prepared for download

Technical Formulas

Disk Performance Formulas (sec-2-1.pdf)

1. **Service Time (Ts) Calculation**

The service time represents the average time for a single I/O operation:

$$Ts = seek\_time + rotational\_latency + transfer\_time$$

Formula Components:

$$T_s = t_{seek} + \frac{0.5 \times 60000}{RPM} + \frac{block\_size}{transfer\_rate}$$

Where:

- t_{seek} = Average disk seek time in milliseconds
- RPM = Disk rotational speed (revolutions per minute)
- $block\_size$ = Data block size in kilobytes
- $transfer\_rate$ = Disk transfer rate in MB/s

Code Implementation:

def calculate_service_time(seek_time_ms, disk_rpm, data_block_size_kb, transfer_rate_mbps):
"""
Calculate average disk service time using formula from sec-2-1.pdf

```
Args:
    seek_time_ms: Average seek time in milliseconds
    disk_rpm: Disk rotation speed in RPM
    data_block_size_kb: Data block size in kilobytes
    transfer_rate_mbps: Transfer rate in MB/s

Returns:
    Service time in milliseconds
"""
# Rotational latency: average time for platter rotation
rotational_latency = (0.5 / (disk_rpm / 60000))

# Transfer time: time to transfer block
transfer_time = (data_block_size_kb / transfer_rate_mbps)

# Total service time
total_time = seek_time_ms + rotational_latency + transfer_time

return total_time
```

Example:
For a 15K RPM disk with 5ms seek time, 4KB block, 40MB/s transfer rate:

$$Ts = 5.0 + \frac{0.5 \times 60000}{15000} + \frac{4}{40} = 5.0 + 2.0 + 0.1 = 7.1 \text{ ms}$$

---

2. IOPS Calculation

IOPS (Input/Output Operations Per Second) represents how many I/O operations a disk can handle:

$$IOPS = \frac{1}{T_s \times 0.001}$$

For 70% Utilization (recommended):

$$IOPS_{70\%} = 0.7 \times \frac{1}{T_s \times 0.001}$$

Code Implementation:

def calculate_iops(service_time_ms, utilization=1.0):
"""
Calculate IOPS from service time

```
Args:
    service_time_ms: Service time in milliseconds
    utilization: Disk utilization factor (0.0 to 1.0)
                Recommended: 0.7 for acceptable response time

Returns:
    IOPS value
"""
iops = utilization / (service_time_ms * 0.001)
return iops
```

Example:
With service time of 7.1ms:

- 100% utilization: IOPS = 1/(7.1×0.001) = 140.8 IOPS
- 70% utilization: IOPS = 0.7 × 140.8 = 98.6 IOPS (recommended)

---

3. Disk Capacity Required (Dc)

The number of disks required to store the total capacity:

$$D_c = \lceil \frac{Total\_Capacity}{Single\_Disk\_Capacity} \rceil$$

Code Implementation:

def calculate_disk_capacity_required(total_capacity_gb, disk_capacity_gb):
"""
Calculate number of disks required for total capacity

```
Args:
    total_capacity_gb: Total storage needed in GB
    disk_capacity_gb: Single disk capacity in GB

Returns:
    Number of disks required (ceiling)
"""
import math
return math.ceil(total_capacity_gb / disk_capacity_gb)
```

---

4. Disk Performance Required (Dp)

The number of disks required to handle the IOPS workload:

$$D_p = \lceil \frac{Application\_IOPS}{Single\_Disk\_IOPS} \rceil$$

Code Implementation:

def calculate_disk_performance_required(app_iops, disk_iops):
"""
Calculate number of disks required for performance

```
Args:
    app_iops: IOPS required by application
    disk_iops: IOPS capacity of single disk (at 70% utilization)

Returns:
    Number of disks required (ceiling)
"""
```

```
import math
return math.ceil(app_iops / disk_iops)
```

---

5. Total Disks Required

The final disk requirement is the maximum of capacity and performance needs:

$$D_{total} = \max(D_c, D_p)$$

The bottleneck is identified by which requirement is higher.

RAID Calculation Formulas (sec-5-1.pdf)

1. Usable Capacity

The usable storage capacity varies by RAID level:

RAID 0 (Striping):

$$C_{usable} = n \times C_{disk}$$

(100% of total capacity)

RAID 1 (Mirroring):

$$C_{usable} = \frac{n}{2} \times C_{disk}$$

(50% of total capacity)

RAID 5 (Striping with Parity):

$$C_{usable} = (n - 1) \times C_{disk}$$

where n is number of disks

Code Implementation:

def usable_capacity_percent(num_disks, raid_level):
"""
Calculate usable capacity percentage for RAID level

```
Args:
    num_disks: Number of disks in array
    raid_level: "RAID 0", "RAID 1", or "RAID 5"

Returns:
    Usable capacity as percentage (0-100)
"""
if raid_level == "RAID 0":
    return 100.0
```

```
elif raid_level == "RAID 1":
    return 50.0
elif raid_level == "RAID 5":
    return ((num_disks - 1) / num_disks) * 100
else:
    return 0.0
```

---

2. Write Penalty

RAID levels incur different overhead for write operations:

RAID Level	Write Penalty	Reason
RAID 0	1	No redundancy overhead
RAID 1	2	Write to both mirrors
RAID 5	4	Read data + read parity + write data + write parity

Code Implementation:

def get_write_penalty(raid_level):
"""
Get write penalty multiplier for RAID level

```
Args:
    raid_level: "RAID 0", "RAID 1", or "RAID 5"

Returns:
    Write penalty factor
"""
penalties = {
    "RAID 0": 1,
    "RAID 1": 2,
    "RAID 5": 4
}
return penalties.get(raid_level, 1)
```

---

3. Disk Load with Write Penalty

The actual load on disks considering the workload mix:

$$DiskLoad = (Total\_IOPS \times Read\%) + (Total\_IOPS \times Write\% \times WritePenalty)$$

Code Implementation:

def calculate_disk_load_iops(total_iops, read_percent, write_percent, raid_level):
"""
Calculate actual disk load considering write penalty

```
Args:
    total_iops: Total IOPS from application
    read_percent: Read operations percentage (0-100)
    write_percent: Write operations percentage (0-100)
    raid_level: "RAID 0", "RAID 1", or "RAID 5"

Returns:
    Actual disk load in IOPS
"""
write_penalty = get_write_penalty(raid_level)
read_load = total_iops * (read_percent / 100)
write_load = total_iops * (write_percent / 100) * write_penalty

return read_load + write_load
```

Example:
For 400 IOPS with 75% read, 25% write on RAID 5:

- Read load: 400 × 0.75 = 300 IOPS
- Write load: 400 × 0.25 × 4 = 400 IOPS
- Total disk load: 700 IOPS

Module Documentation

run.py - Main Application (1,200+ lines)

The main application file orchestrates the entire workflow:

Key Sections:

1. **Import Statements** - Required libraries and custom modules
2. **Path Configuration** - Setup for reports directory and module paths
3. **Global Constants** - Supported file types, disk specifications
4. **Folder Scanning** - Recursive file discovery and metadata collection

5. **Disk Performance Calculations** - Implements formulas from sec-2-1.pdf
6. **Virtual Disk Builder** - Creates simulated disk structures
7. **Main Simulation Runner** - Orchestrates complete simulation workflow
8. **Gradio UI** - Interactive web interface with three main tabs

Code Structure:

Section 1: Imports

```
import gradio as gr
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
```

Section 2: Path Configuration

```
APP_DIR = Path(file).parent
REPORTS_DIR = APP_DIR / "reports"
REPORTS_DIR.mkdir(exist_ok=True)
```

Section 3: Import Custom Modules

```
from Data_Analysis import append_run, save_report_csv
from Raid_Calculation import usable_capacity_percent, space_efficiency
from Raid_Simulation import simulate_raid_read, simulate_raid_write
from Disk_Performance import calculate_service_time, calculate_iops
```

Section 4: Global Constants

```
SUPPORTED_EXT = {".jpg", ".jpeg", ".png", ".bmp", ".gif", ".mp4", ".mov", ".mkv"}
DISK_SPECS = {
"seek_time_ms": 5.0,
"rpm": 15000,
"transfer_rate_mbps": 40,
"capacity_gb": 100,
"io_size_kb": 4
}
```

Section 5-9: Function definitions...

Section 10: Gradio UI setup...

Disk_Performance Module

Implements all disk performance formulas from sec-2-1.pdf:

Key Functions:

1. `calculate_service_time()` - Computes Ts
2. `calculate_iops()` - Computes IOPS from Ts
3. `calculate_rotational_latency()` - Rotational latency component
4. `calculate_disk_capacity_required()` - Computes Dc
5. `calculate_disk_performance_required()` - Computes Dp
6. `calculate_total_disks_required()` - Computes total with bottleneck analysis
7. `complete_disk_analysis()` - Full pipeline

Raid_Calculation Module

Implements all RAID formulas from sec-5-1.pdf:

Key Functions:

1. `usable_capacity_percent()` - Capacity by RAID level
2. `redundancy_percent()` - Redundancy overhead
3. `space_efficiency()` - Storage efficiency ratio
4. `get_write_penalty()` - Write penalty by RAID level
5. `calculate_disk_load_iops()` - Disk load with write penalty
6. `calculate_capacity_breakdown_dict()` - Detailed capacity breakdown
7. `calculate_xor_parity()` - XOR parity calculation

Raid_Simulation Module

Simulates read/write performance:

Key Functions:

1. `simulate_raid_read()` - Simulates read operation timing
2. `simulate_raid_write()` - Simulates write with penalty
3. `calculate_raid_iops()` - Array IOPS capacity

Data_Analysis Module

Handles data collection and reporting:

Key Functions:

1. `append_run()` - Logs simulation run to DataFrame
2. `save_report_csv()` - Exports to CSV
3. `summary_statistics()` - Computes statistics (mean, median, std, min, max)

Installation and Setup

Prerequisites

- Python 3.7 or higher
- pip (Python package installer)

Installation Steps

Step 1: Install Dependencies

```
pip install gradio pandas matplotlib pillow
```

Step 2: Clone Repository

```
git clone https://github.com/sefffo/Information-Storage-Project.git
cd Information-Storage-Project
```

Step 3: Verify Installation

```
python -c "import gradio; import pandas; print('All dependencies installed!')"
```

Step 4: Run Application

```
python run.py
```

The application will automatically launch in your default web browser at `http://127.0.0.1:7860`

Usage Guide

Step 1: Load Media Files

1. Click the **"1. Load"** tab
2. Enter the path to a folder containing multimedia files
    - Example: `"C:\Users\YourName\Pictures"` (Windows)
    - Example: `"/home/username/Pictures"` (Linux/Mac)
3. Click the **"Scan"** button
4. Review the scan results showing:
    - Total number of files
    - Total size in MB
    - File types distribution
    - Detailed file table with sizes and resolutions

Supported File Types:

- Images: `.jpg`, `.jpeg`, `.png`, `.bmp`, `.gif`
- Videos: `.mp4`, `.mov`, `.mkv`, `.avi`, `.webm`

Step 2: Configure RAID Settings

1. Click the **"2. Config"** tab
2. **Select number of disks** using the slider (2-12 disks)
3. **Select RAID level** from radio buttons:
    - **RAID 0**: Maximum performance, no redundancy
    - **RAID 1**: Full redundancy, 50% usable capacity
    - **RAID 5**: Balanced redundancy, (n-1)/n usable capacity
4. Click **"Calc Metrics"** to preview theoretical metrics
5. Review displayed calculations:
    - Usable capacity percentage
    - Space efficiency ratio
    - Write penalty factor

Step 3: Run Simulation

1. Click the **"3. Run"** tab
2. Click **" Simulate"** button to execute simulation
3. Wait for simulation to complete (typically 5-30 seconds)
4. Review results:
    - **Disk Performance Metrics**: Service time, IOPS, bottleneck analysis
    - **Capacity Distribution Chart**: Pie chart showing usable/parity/mirror
    - **Performance Chart**: Bar chart showing total operation time
    - **Run Data Table**: Complete metrics from simulation
    - **Summary Statistics**: Statistical analysis (mean, median, std, min, max)
5. **Download Results**: Click file download button to download:
    - CSV report files
    - PNG chart visualizations
    - Virtual disks ZIP archive

Code Snippets

Snippet 1: Complete Scan and Analysis

def scan_folder(folder_path):
"""
Complete folder scanning workflow
"""
# Normalize path
folder_path = os.path.normpath(folder_path.strip('"').strip("'"))

```
# Validate folder exists
if not os.path.isdir(folder_path):
    return "✖ Invalid folder.", None, 0

# Find all multimedia files
files = [
    os.path.join(r, f)
    for r, _, n in os.walk(folder_path)
    for f in n
    if os.path.splitext(f)[1].lower() in SUPPORTED_EXT
]

if not files:
    return "✖ No media files found.", None, 0

# Analyze each file
rows = []
total_bytes = 0

for path in files:
    size = os.path.getsize(path)
    total_bytes += size
    resolution = "N/A"

    # Extract image resolution if applicable
    if path.lower().endswith(('.jpg', '.png', '.bmp', '.gif')):
        try:
            with Image.open(path) as img:
                resolution = f"{img.width}x{img.height}"
        except:
            pass

    rows.append({
        "Filename": os.path.basename(path),
        "Extension": os.path.splitext(path)[1],
        "Size (KB)": round(size/1024, 2),
        "Resolution": resolution
    })

# Store results globally
last_scan.update({
    "folder": folder_path,
    "files": files,
    "total_bytes": total_bytes
})
```

```
# Generate summary
df = pd.DataFrame(rows)
ext_counts = df["Extension"].value_counts().to_string()
summary = f"""
## 󰄵 Scan Results
- **Files:** {len(files)}
- **Size:** {total_bytes/1024**2:.2f} MB

### Types:
{ext_counts}
"""

return summary, df[:200], total_bytes/1024**2
```

Snippet 2: Disk Performance Calculation Pipeline

def calculate_disk_metrics(num_disks, raid_level, total_capacity_gb, app_iops_required):
"""
Complete disk performance analysis using formulas from sec-2-1.pdf
"""
# Step 1: Calculate Service Time (Ts)
service_time = calculate_service_time(
seek_time_ms=DISK_SPECS["seek_time_ms"],
disk_rpm=DISK_SPECS["rpm"],
data_block_size_kb=DISK_SPECS["io_size_kb"],
transfer_rate_mbps=DISK_SPECS["transfer_rate_mbps"]
)

```
# Step 2: Calculate IOPS per disk (70% utilization for acceptable response)
iops_per_disk = calculate_iops(service_time, utilization=0.7)

# Step 3: Calculate Dc (disks for capacity)
disks_for_capacity = calculate_disk_capacity_required(
    total_capacity_gb,
    DISK_SPECS["capacity_gb"]
)

# Step 4: Calculate Dp (disks for performance)
disks_for_performance = calculate_disk_performance_required(
    app_iops_required,
    iops_per_disk
)

# Step 5: Determine total and bottleneck
total_disks = max(disks_for_capacity, disks_for_performance)
is_capacity_bottleneck = disks_for_capacity > disks_for_performance

return {
    "service_time_ms": service_time,
    "iops_per_disk": iops_per_disk,
    "disks_for_capacity": disks_for_capacity,
    "disks_for_performance": disks_for_performance,
    "total_disks_required": total_disks,
    "is_capacity_bottleneck": is_capacity_bottleneck
}
```

Snippet 3: RAID IOPS with Write Penalty

def calculate_raid_iops_with_workload(num_disks, raid_level, iops_per_disk, read_percent=70):
"""
Calculate RAID array IOPS considering write penalty (sec-5-1.pdf)
"""
write_percent = 100 - read_percent

```
# Maximum IOPS the array can theoretically provide
if raid_level == "RAID 0":
    max_array_iops = iops_per_disk * num_disks
elif raid_level == "RAID 1":
    max_array_iops = iops_per_disk * num_disks * 0.6
elif raid_level == "RAID 5":
    max_array_iops = iops_per_disk * (num_disks - 1) * 0.7
else:
    max_array_iops = iops_per_disk * num_disks

# Calculate disk load for typical workload
disk_load = calculate_disk_load_iops(
    total_iops=max_array_iops,
    read_percent=read_percent,
    write_percent=write_percent,
    raid_level=raid_level
)

write_penalty = get_write_penalty(raid_level)

return {
    "total_iops_capacity": int(max_array_iops),
    "disk_load_iops": int(disk_load),
    "write_penalty": write_penalty
}
```

Snippet 4: Virtual Disk Structure Creation

def build_virtual_disks(raid, num_disks, files, timestamp):
"""
Create virtual disk structure simulating actual RAID data distribution
"""
# Create directory structure for virtual disks
base_path = REPORTS_DIR / f"virtual_disks_{timestamp}" / raid.replace(" ", "*")
disks = *{i: base_path / f"disk*{i}" for i in range(num_disks)}

```
for disk_path in disks.values():
    disk_path.mkdir(parents=True, exist_ok=True)

# Track disk loads for load balancing
disk_loads = {i: 0 for i in range(num_disks)}

# Distribute files across disks
for idx, file_path in enumerate(files):
    size = os.path.getsize(file_path)

    if raid == "RAID 1":
        # Mirror: copy to all disks
        targets = list(disks.keys())
    else:
        # RAID 0/5: distribute based on load
        candidates = [
            d for d in range(num_disks)
            if (raid != "RAID 5" or d != (idx % num_disks))
        ]
        target = min(candidates, key=lambda x: disk_loads[x])
        targets = [target]

    # Copy file to target disks
    for target_disk in targets:
        shutil.copy2(file_path, disks[target_disk])
        disk_loads[target_disk] += size

    # For RAID 5, create parity block
    if raid == "RAID 5":
        parity_disk = idx % num_disks
        parity_file = disks[parity_disk] / f"{Path(file_path).stem}_PARITY.txt"
        parity_file.write_text(f"Parity for {Path(file_path).name}\nOn disk {targets[(

# Create ZIP archive of virtual disks
zip_path = shutil.make_archive(
    str(base_path.parent / base_path.name),
    "zip",
    base_path
)

return str(base_path), zip_path
```

Snippet 5: Main Simulation Runner

def run_simulation(num_disks, raid_level):
"""
Execute complete RAID simulation with all calculations
"""
if not last_scan["files"]:
return "❌ Scan folder first.", *([None]*5)

```
# Calculate total capacity in GB
total_capacity_gb = last_scan["total_bytes"] / (1024**3)

# Estimate application IOPS (10 IOPS per 100 files)
```

```
estimated_app_iops = max(100, len(last_scan["files"]) * 0.1)

# Get disk performance metrics
disk_metrics = calculate_disk_metrics(
    num_disks,
    raid_level,
    total_capacity_gb,
    estimated_app_iops
)

# Get RAID IOPS with write penalty
raid_iops = calculate_raid_iops_with_workload(
    num_disks,
    raid_level,
    disk_metrics["iops_per_disk"]
)

# Simulate read/write performance
read_ms = simulate_raid_read(raid_level, num_disks)
write_ms = simulate_raid_write(raid_level, num_disks)

# Collect all metrics
run_data = {
    "raid_level": raid_level,
    "disk_count": num_disks,
    "total_files": len(last_scan["files"]),
    "total_size_gb": total_capacity_gb,
    "read_time_ms": read_ms,
    "write_time_ms": write_ms,
    "total_time_ms": read_ms + write_ms,
    "service_time_ms": disk_metrics["service_time_ms"],
    "iops_per_disk": int(disk_metrics["iops_per_disk"]),
    "disks_for_capacity_Dc": disk_metrics["disks_for_capacity"],
    "disks_for_performance_Dp": disk_metrics["disks_for_performance"],
    "total_disks_required": disk_metrics["total_disks_required"],
    "array_iops_capacity": raid_iops["total_iops_capacity"],
    "write_penalty": raid_iops["write_penalty"],
    "usable_%": usable_capacity_percent(num_disks, raid_level),
    "efficiency_%": space_efficiency(num_disks, raid_level) * 100
}

# ... rest of simulation ...
return results
```

Performance Metrics

Disk Performance Analysis

The simulator calculates key disk performance metrics:

Service Time (Ts)

- Typical range: 5-15 ms for enterprise disks
- Components: seek time + rotational latency + transfer time
- Impact: Lower values allow higher IOPS

IOPS (Input/Output Operations Per Second)

- Calculation: 1 / (Ts in seconds)
- 100% utilization: Theoretical maximum
- 70% utilization: Practical target for acceptable response times
- Example: 7.1ms service time = 98 IOPS at 70% utilization

Disk Requirements

Metric	Meaning	Impact
Dc (Capacity)	Disks needed for storage capacity	Minimum disks required
Dp (Performance)	Disks needed for IOPS workload	Performance constraint
Bottleneck	Which requirement is limiting	Design bottleneck

RAID Performance Comparison

RAID 0 (Striping)

- Read Performance: Maximum (all disks participate)
- Write Performance: Maximum (all disks participate)
- Fault Tolerance: None
- Usable Capacity: 100%
- Write Penalty: 1x

RAID 1 (Mirroring)

- Read Performance: High (read from either disk)
- Write Performance: Moderate (write to both disks)
- Fault Tolerance: Can survive 1 disk failure per mirror pair
- Usable Capacity: 50%
- Write Penalty: 2x

RAID 5 (Striping with Parity)

- Read Performance: High (parallel reads from multiple disks)
- Write Performance: Moderate (parity calculation overhead)
- Fault Tolerance: Can survive 1 disk failure
- Usable Capacity: (n-1)/n of total (e.g., 80% for 5 disks)
- Write Penalty: 4x (read data, read parity, write data, write parity)

Results and Visualization

Output Files Generated

Each simulation generates comprehensive results:

CSV Reports:

1. **report_[timestamp].csv** - Detailed simulation metrics
2. **summary_[timestamp].csv** - Statistical summary (mean, median, std, min, max)

Visualizations:

1. **pie_[timestamp].png** - Capacity distribution pie chart
2. **bar_[timestamp].png** - Performance metrics bar chart

Virtual Disks:

1. **virtual_disks_[timestamp].zip** - Simulated disk file distribution

Sample Output

Capacity Distribution Chart:

- Usable: 80% (RAID 5 with 5 disks)
- Parity: 20% (dedicated to redundancy)

Performance Report:

- Service Time: 7.10 ms
- IOPS per Disk: 98 IOPS (70% utilization)
- Array IOPS Capacity: 392 IOPS (RAID 5, 5 disks)
- Write Penalty: 4x
- Total Operation Time: 2.8 seconds (1.23s read + 1.58s write)

Troubleshooting

Issue: "No media files found"

Solution:

- Verify folder path is correct
- Ensure folder contains supported file types:
  - Images: .jpg, .jpeg, .png, .bmp, .gif
  - Videos: .mp4, .mov, .mkv, .avi, .webm
- Check file extensions are lowercase (or update SUPPORTED_EXT)
- Ensure read permissions on folder

Issue: Simulation takes too long

Solution:

- Reduce number of files by scanning smaller folder
- Reduce number of disks (less virtual structure to create)
- Close other applications to free system resources

Issue: "Invalid folder" error

Solution:

- Remove quotes from folder path
- Use forward slashes (/) on Windows or backslashes (\)
- Verify folder exists with valid permissions
- Use absolute path instead of relative path

Issue: Charts not displaying

Solution:

- Ensure matplotlib is installed: `pip install matplotlib`
- Check disk space available (reports directory)
- Verify write permissions in reports folder

Advanced Configuration

Customizing Disk Specifications

Edit `DISK_SPECS` dictionary in `run.py`:

```
DISK_SPECS = {
"seek_time_ms": 5.0, # Average seek time
"rpm": 15000, # RPM (15K for enterprise, 7.2K for consumer)
"transfer_rate_mbps": 40, # MB/s transfer rate
"capacity_gb": 100, # Single disk capacity
"io_size_kb": 4 # Typical I/O block size
}
```

Changing Supported File Types

Edit SUPPORTED_EXT set in run.py:

```
SUPPORTED_EXT = {
# Add or remove file extensions
".jpg", ".jpeg", ".png", # Image formats
".mp4", ".mov", ".mkv", # Video formats
".flac", ".wav", ".mp3" # Audio formats (if desired)
}
```

Conclusion

This RAID Multimedia Storage Performance Simulator demonstrates the integration of theoretical concepts from information storage systems with practical implementation. The project successfully implements:

- Disk performance calculations based on sec-2-1.pdf formulas
- RAID capacity and efficiency calculations based on sec-5-1.pdf formulas
- Interactive visualization of storage system design decisions
- Educational simulation of real-world RAID configurations

The system provides valuable insights into:

- Trade-offs between capacity, performance, and redundancy
- Impact of write operations on system design
- Relationship between disk specifications and overall system performance
- Practical constraints in storage system design

References

1. **sec-2-1.pdf** - Data Center Environment (Disk Performance Formulas)
2. **sec-5-1.pdf** - RAID Techniques (RAID Capacity and Write Penalty Formulas)
3. Gradio Documentation: https://gradio.app/
4. Pandas Documentation: https://pandas.pydata.org/
5. Matplotlib Documentation: https://matplotlib.org/

Project Repository: https://github.com/sefffo/Information-Storage-Project

Last Updated: December 2025

Documentation Version: 2.0