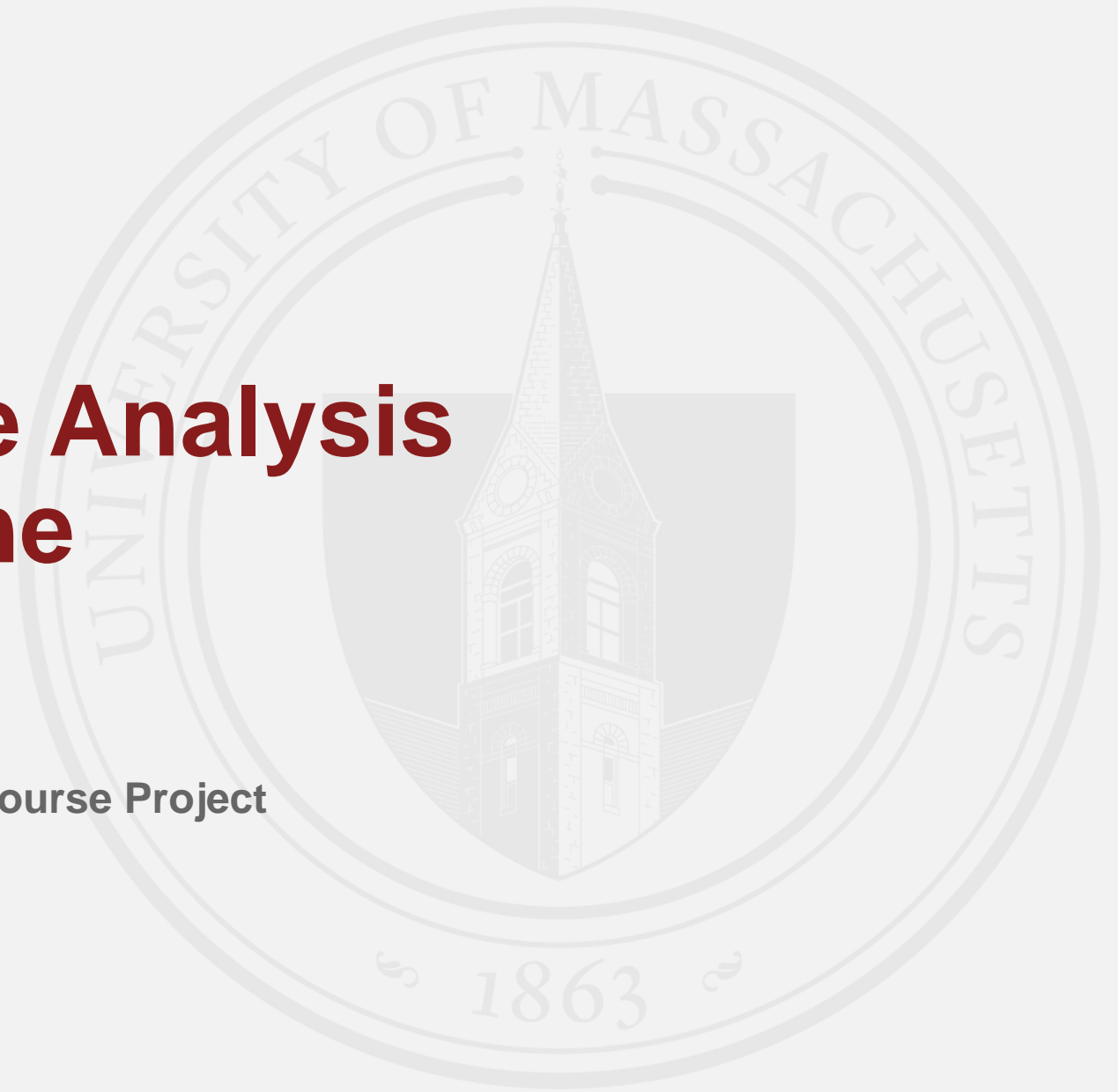


System Performance Analysis on IMDb Data Pipeline

Systems for Data Science COMPSCI 532 SP25 Course Project

Team Member: Ash Pai, Shishira Das, Yajie Liu

Date: May 2025



Contents

- 1 Project Overview**
- 2 Project Goals**
- 3 System Design & Architecture**
- 4 Challenges & Solutions**
- 5 Performance Evaluation**
- 6 Summary & Possible Extensions**

Project Overview



Background

- Modern applications generate massive amounts of data, making efficient data processing and querying critical challenges.
- Managing large and complex datasets may encounter:
 - Performance bottlenecks
 - Scalability limitations

Motivation

- To build a system for analyzing and optimizing query performance using the IMDb dataset.
- To understand how different constraints and system designs impact query efficiency.
- To gain practical insights into optimization techniques for large-scale data systems.

IMDb Dataset

This dataset is sourced from IMDb, the world's most popular and authoritative source for movie, TV, and celebrity content. It includes extensive details on movies, **TV series, episodes, ratings, crew members, and more.**

Available Files & Their Contents:

1. **title.akas.tsv.gz** – Alternative titles, languages, and regional information.
2. **title.basics.tsv.gz** – Movie and TV series metadata (title, year, genre, runtime, etc.).
3. **title.crew.tsv.gz** – Directors and writers for each title
4. **title.episode.tsv.gz** – Episode details (season, episode number, parent series)
5. **title.principals.tsv.gz** – Key cast and crew members
6. **title.ratings.tsv.gz** – IMDb ratings and vote counts
7. **name.basics.tsv.gz** – Actor and filmmaker information (birth year, known titles, professions)

License & Usage

This dataset is provided for **personal and non-commercial** use under IMDb's terms and conditions. If you use this data, you must credit IMDb as the source. Please refer to the Non-Commercial Licensing and copyright/license and verify compliance.

Acknowledgments

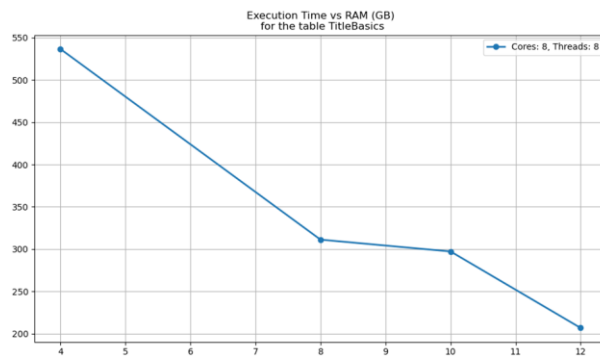
This dataset is officially provided by IMDb and curated for easy access.

Project Goals

1

Performance benchmarking based on different combinations of hardware:

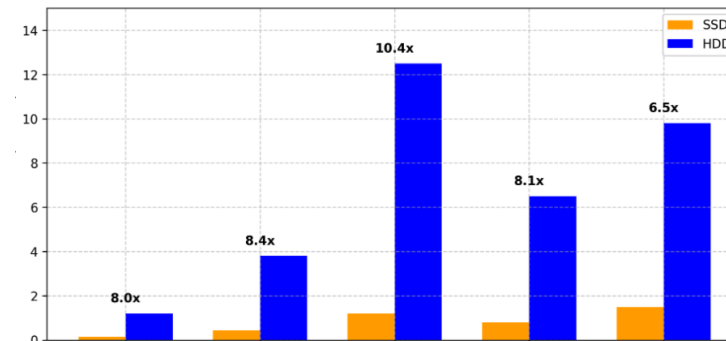
- a. Number of CPU Cores
- b. Amount of RAM
- c. Number of Threads



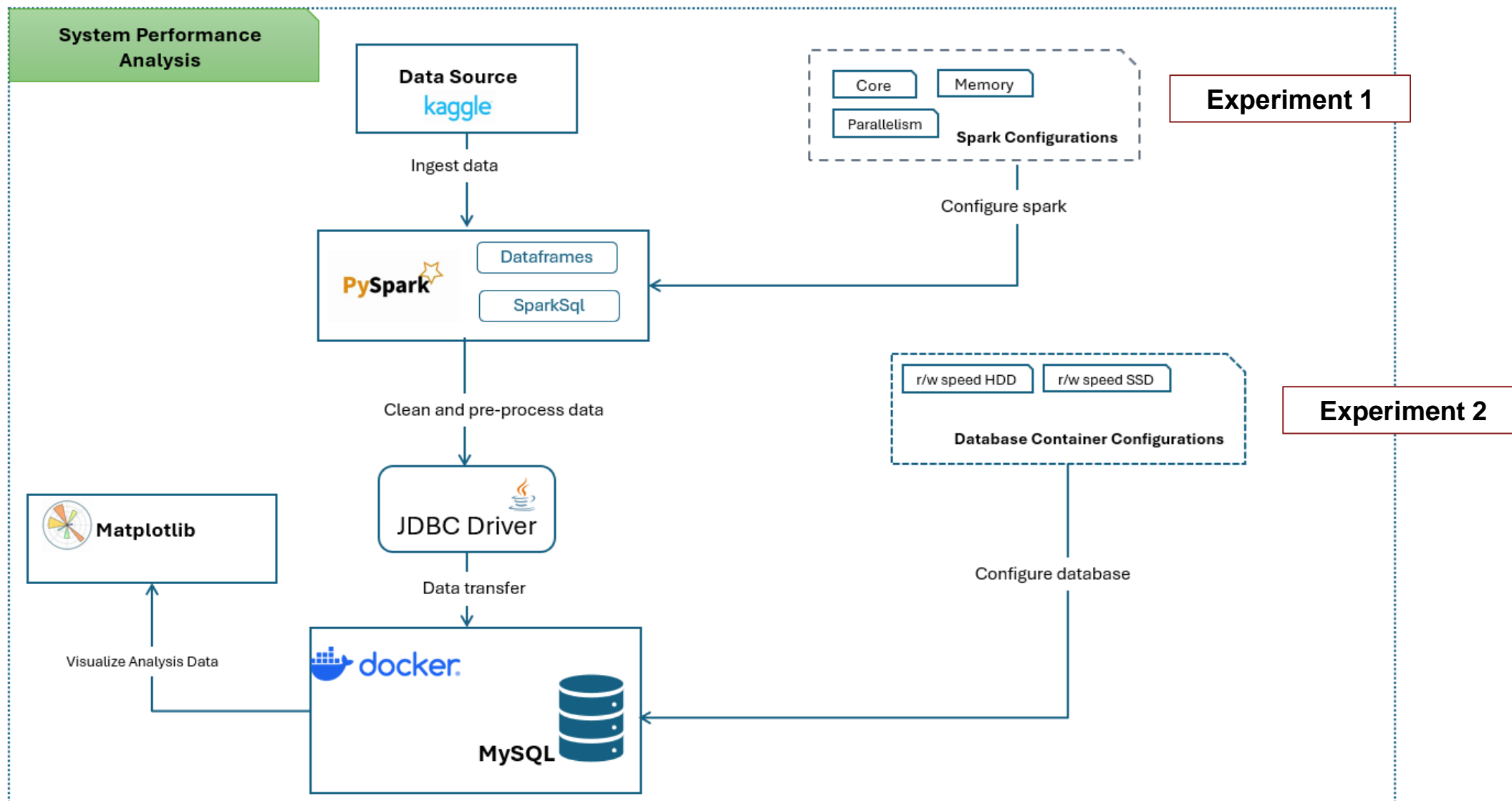
2

Query processing performance analysis under different disk configurations:

- a. SSD
- b. HDD



System Design & Architecture



Key Design Decisions:

1. Modular Architecture:

Our project has a modular architecture pattern with clearly separated components, each with distinct responsibilities:

Data Download: Handles authentication with Kaggle API, manages download requests, and ensures data integrity during acquisition

Data Cleaning: Transforms raw data through a series of specialized cleaning functions, each tailored to specific IMDb datasets

Database Interaction: Manages connection pooling, schema creation, and data insertion into MySQL

Spark Performance Testing: Collects and visualizes performance metrics across different system configurations while processing different data cleaning and data ingestion operations

MySQL Query Processing Testing: Performs basic queries to measure performance differences when simulating SSD and HDD technologies.

2. Scalable Processing:

We selected **PySpark** for batch data processing due to its advantages over other approaches like MapReduce.

Batch Ingestion: PySpark handles the entire IMDb dataset (multi-GB) in memory rather than through disk-based processing stages, resulting in significant performance improvements.

Declarative Data Transformations: Our cleaning functions use a chain of DataFrame operations, automatically optimized by Spark's query planner.

In-Memory Computation and Lazy Evaluation: PySpark caches intermediate results in memory during batch processing, avoiding costly disk I/O operations that would bottleneck traditional MapReduce approaches. It also invokes lazy evaluation, allowing Spark to optimize the entire processing plan before execution.

3. Containerized Storage:

MySQL in Docker for isolated, reproducible database environment, with below advantages:

Environment Isolation: The database environment is completely isolated from the host system, preventing conflicts with other services

Reusable: The container configuration is version-controlled, ensuring consistent database setups across development and testing

Resource Configurable: We can explicitly limit CPU, memory, and I/O resources for configurable performance test

Portability: The entire database environment can be deployed consistently across different systems

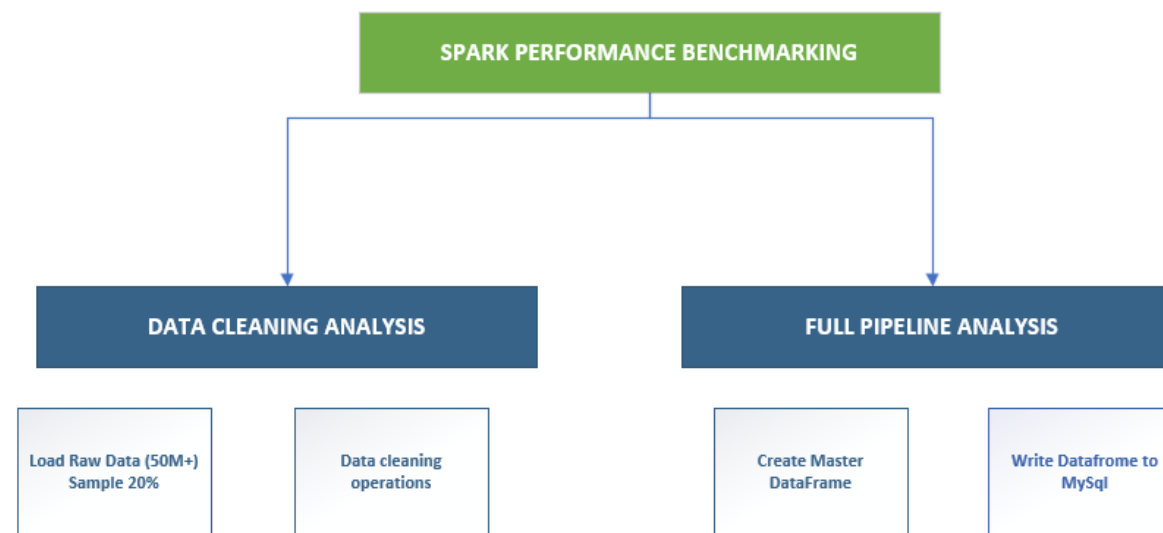
Work Distributions

Task	Team Member(s)
Set up the environment to run a database in Docker to vary the hardware configuration	Ash Pai
Performance benchmarking of batch ingestion algorithm under different hardware constraints	Shishira Das
Performance visualization based on different hardware combinations to identify the most impactful component	Yajie Liu
Analyze database performance differences between SSD and HDD read/write speeds	Ash Pai
Final presentation preparation and video	Ash Pai, Shishira Das, Yajie Liu

Data Cleaning and Database Ingestion : System Performance Analysis



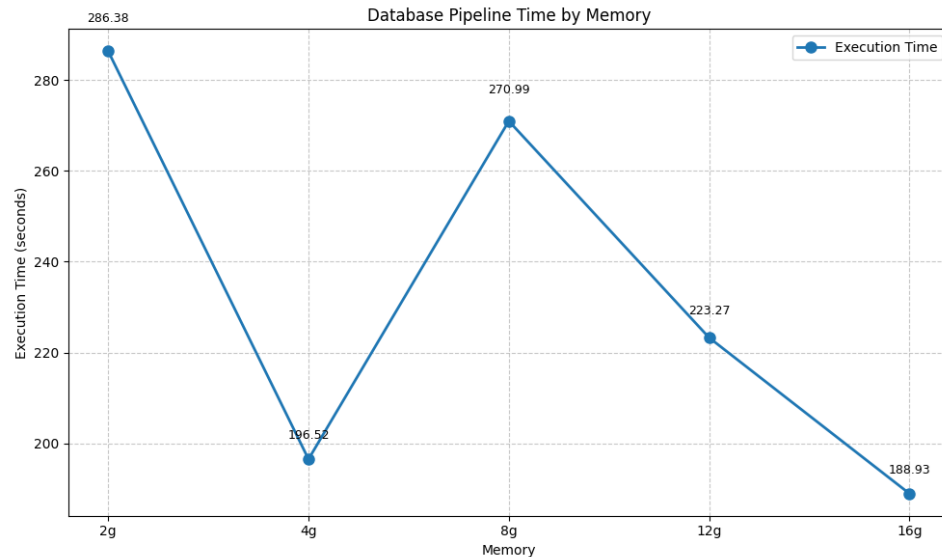
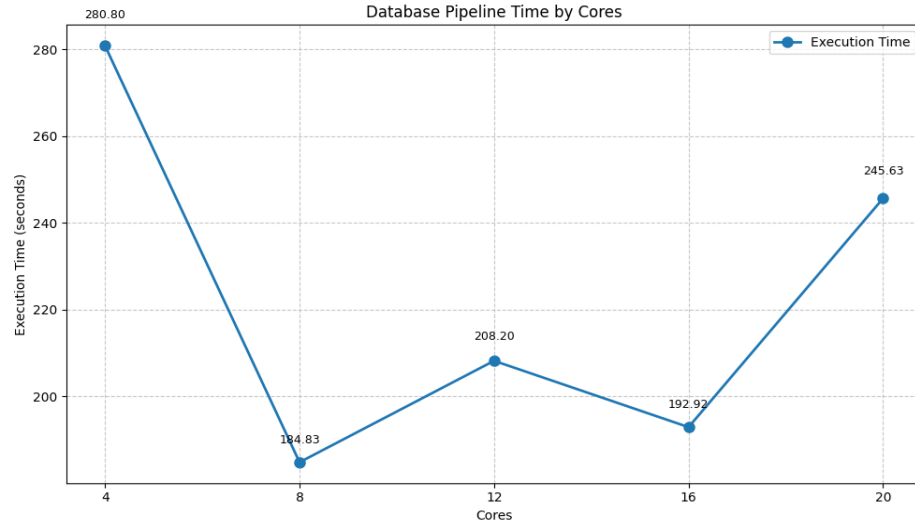
Spark: Performance Benchmarking Framework



Key features:

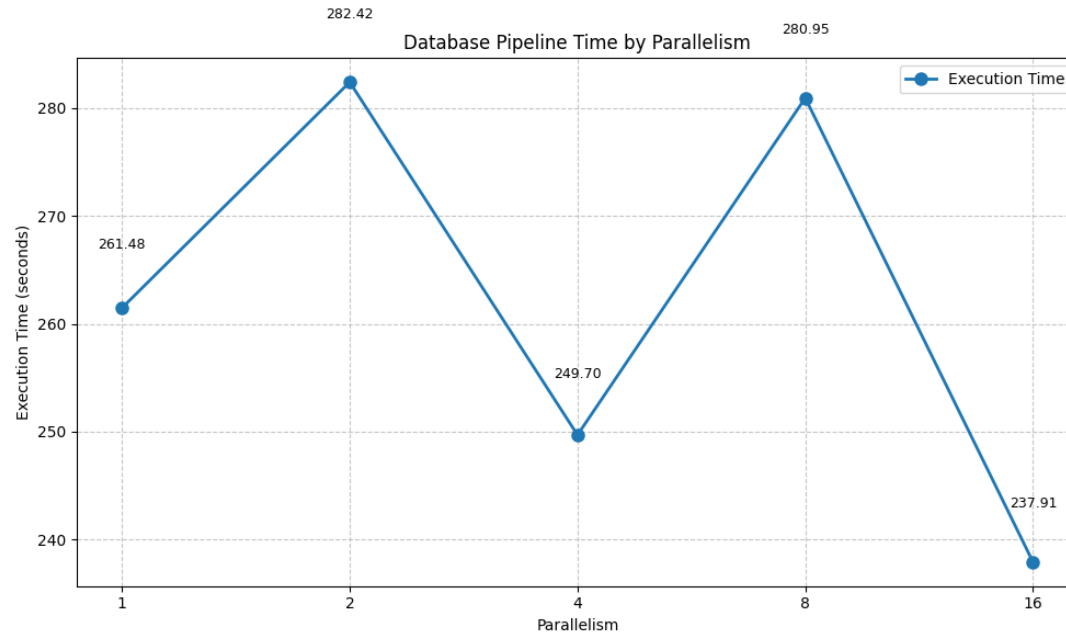
- **Easy Resource Configuration:**
 - We configured Spark sessions with variable system resources (cores, memory, parallelism) through a flexible configuration file, allowing precise control of computational parameters for performance testing.
- **Systematic Parameter Testing with Isolation:**
 - While testing, we evaluate one system parameter at a time while keeping others constant, creating clean isolated session environments for each test to ensure uncontaminated benchmark results.
- **Comprehensive analysis using visual tools**
 - We use Matplotlib to visualize the data as charts showing both processing time and speed (rows/second), helping identify the best system settings for data processing and derive results from it.

Full Data Pipeline Benchmarking



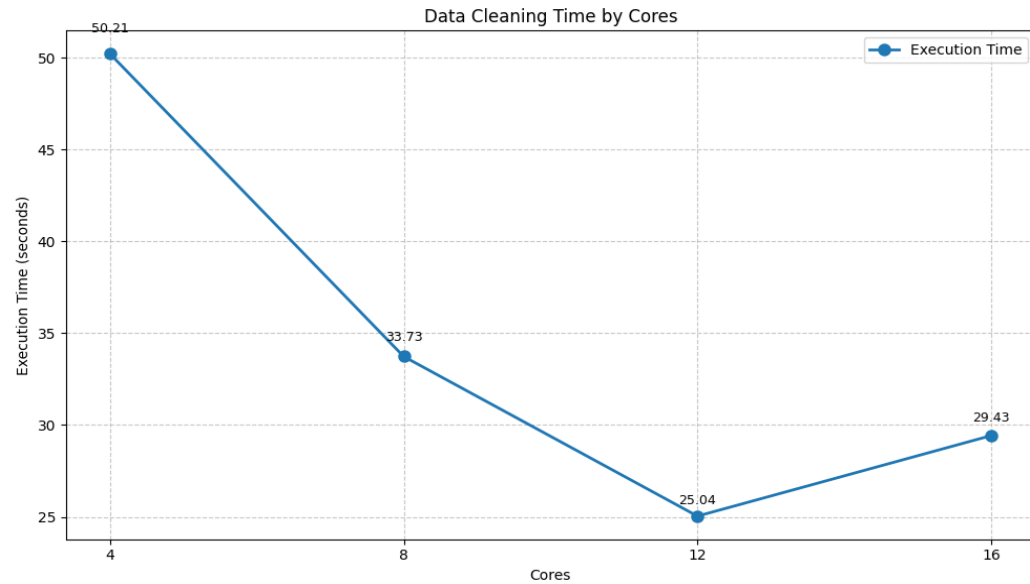
- **Non-Linear Scaling:** Doubling the cores from 8 to 16 does not increase the time linearly, indicating good but not perfect parallelism
- **Diminishing returns:** As we increase the cores, it shows reduced efficiency gains suggesting other bottlenecks. At, 20 cores it increases actually.
- Significant **improvement** when **memory is increased** from 2G to 16G
- This probably indicates that the workload we are testing requires more memory and is not purely computational. The **I/O with Database** could also be a factor along with increasing executor task coordination overhead.

Full Data Pipeline Benchmarking

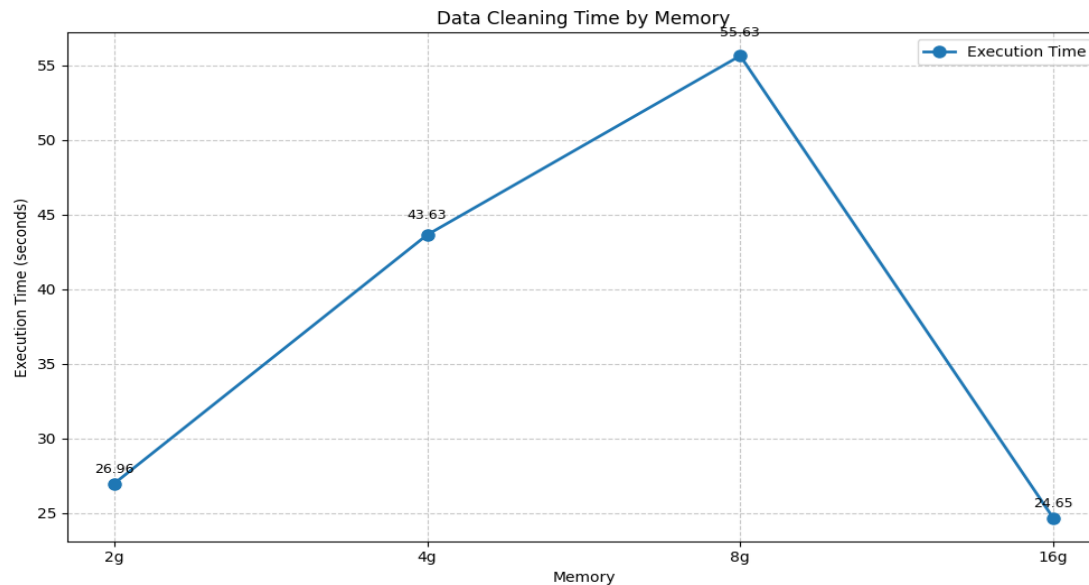


- System generally seems to be **performing better with higher parallelism** at 16 cores per executor but the results are very mixed.
- Better numbers could be due to effective CPU utilization when tasks have wait times due to I/O with the MySQL database
- Overhead of scheduling and context switching in this case seems to have been **offset by the effective resource allocation** at higher core count.
- Optimal configuration for this workload:
 - 8-12 cores
 - 16G memory for optimal performance
 - 16 cores per executor
- Analysis tends to confirm that **performance tuning for Spark operations can be highly dependent on the workload**. Writing to database and dataset size could require high memory and parallelism.

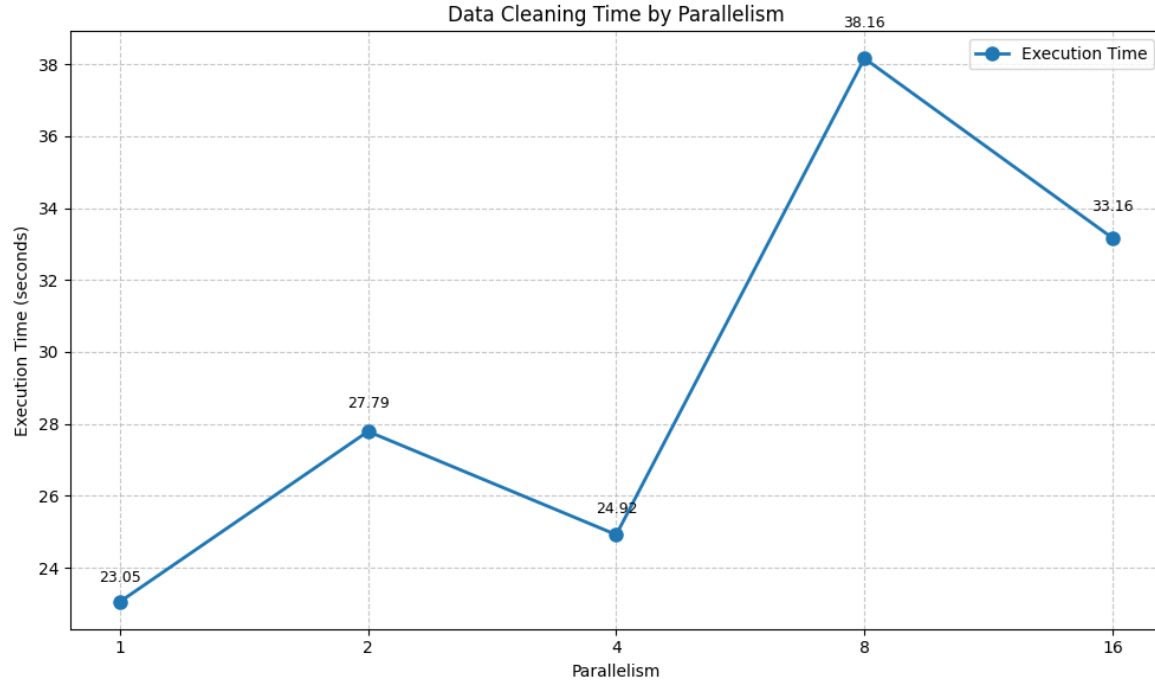
Data Cleaning Benchmarking



- Performance seems to consistently **improve** from **4 to 12 cores**, but degrades at higher value.
- This may suggest that the cleaning operations suffer from context switching and **reach their parallelization limit at 12 cores**
- Interestingly, we see 2g perform better than 8g configuration. This could be due to some caching or memory management done by Spark before utilizing more data in-memory at 16g.
- Spark's Memory configuration and complex caching strategy can impact the tests. It could cause more **disk spilling and GC collection** at lower memory configuration.



Data Cleaning Benchmarking



- **Lower parallelism settings generally outperformed higher ones**, indicating that task coordination overhead outweighs the parallel processing benefits, as seen in CPU tests as well.
- Managing and scheduling many **small tasks reduces overall efficiency**.
- Compared to full data pipeline benchmarking, **cleaning works better with low parallelism**. And adequate memory serves better by consistently improving performance.
- **I/O Bottleneck could have been the limiting factor in full data pipeline analysis** as Spark seems to be doing effective in-memory computation but **gets limited by JDBC Connector**

Spark Performance Benchmarking : Trade Offs



Trade Offs and Challenges:



Parameter Isolation: Varying one parameter at a time gives cleaner results but doesn't reflect real-world multi-parameter optimization



Spark Session Recreation: Ensures proper clean up but adds additional non-measured overhead of start up and data load. Other possible side effects



Performance Testing Range vs Time Taken: Testing more parameter combinations provides better insights but increases testing time exponentially



Sampling vs Accuracy: Enables faster iteration but could miss issues that only appear with full data volume



Environment and Memory Management: Maintaining consistent environment and datasets with 50M+ rows create overload

Validation Tests:

- **Data Integrity Verification:**
 - Verifying row counts between different data pipeline stages
 - Logged top rows and verified data transformations after cleaning steps
- **Benchmark Reliability Tests:**
 - Executed multiple consecutive runs to verify consistent behaviour
 - Adjusted the Spark Configurations so total cores being tested is divisible by the cores being assigned to each executor to avoid wasted resources. Verified this with console logging
- **Integration Testing and Spark Resource Configuration Monitoring:**
 - Database Connector and Storage verified with test connectivity code
 - Resource Configuration verification by logging Spark Session stats
 - Monitoring Runtime metrics on Spark's UI and visualizing DAGs, Stage Job tracking

Known Limitations:

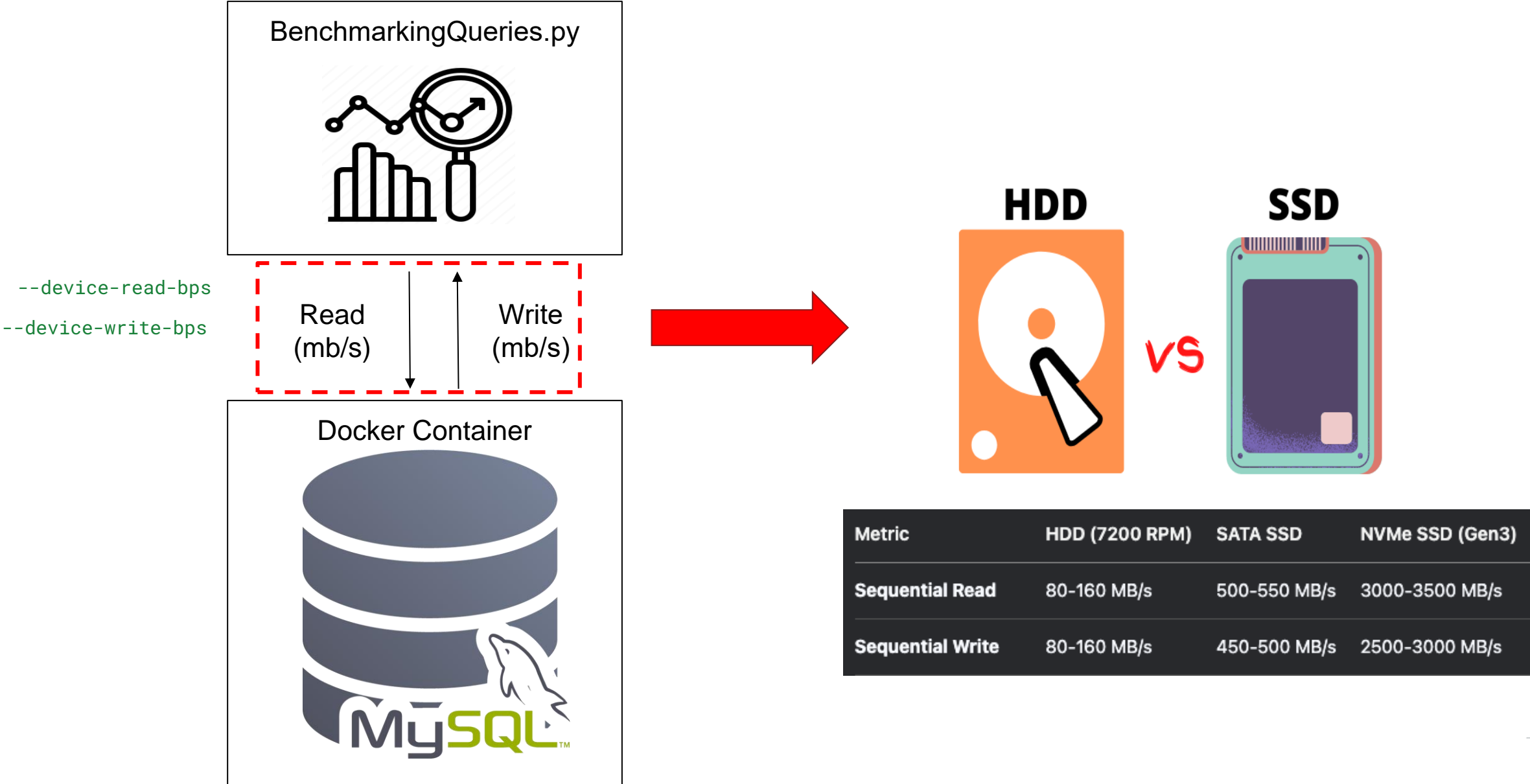
- **Java heap space errors** not always gracefully handled and recovery may require manual intervention.
- Extreme memory limitation can often cause **long running benchmarking and silent errors**.
- Unable to handle large datasets due to hardware constraints and hence limited to sampling metrics
- Inconsistent results across some test runs indicating potential resource contention issues in the VM

Spark Performance Benchmarking: Overall Conclusion

- **More resources don't always mean better performance** or grow linearly and finding a good balance between resources allocation in Spark is critical for cost-effective data processing.
- **Data transformation characteristics** like I/O bound or memory-intensive can heavily influence optimal configurations.
- Generally, **combination of sufficient memory(16G) , adequate cores (12) and moderate parallelism (4) seems to yield better results.**
- The performance pattern demonstrates how **Spark's lazy evaluation and execution planning can optimize operations** when given appropriate resources and configurations.
- In our testing , we got some mixed results, suggesting that there can be **external factors impacting our testing** and Spark's query planner makes different optimization decisions based on available resources.
- There doesn't seem to be any **one-size-fits-all configuration for Spark** and performance tuning is **highly dependent on the workload**. Internal operations like shuffling, caching and task scheduling could come into play when optimizing. This would require more empirical testing with different resource combinations in highly isolated system to get conclusive results for a given workload.

Query Processing Analysis HDD vs. SSD

Query Processing: System Design & Architecture



Testing Queries

```
query_list = [  
    "SELECT COUNT(*) FROM TitleBasics;",  
    "SELECT * FROM TitleRatings WHERE averageRating > 8;",  
    "SELECT t.primaryTitle, r.averageRating FROM TitleBasics t JOIN TitleRatings r ON t.tconst = r.tconst WHERE r.numVotes > 10000;",  
    "SELECT primaryTitle, startYear FROM TitleBasics WHERE startYear BETWEEN 2000 AND 2010 ORDER BY startYear DESC;",  
    "SELECT genres, COUNT(*) as count FROM TitleBasics GROUP BY genres ORDER BY count DESC LIMIT 10;"
```

Container Configurations

Hard Disk Drive

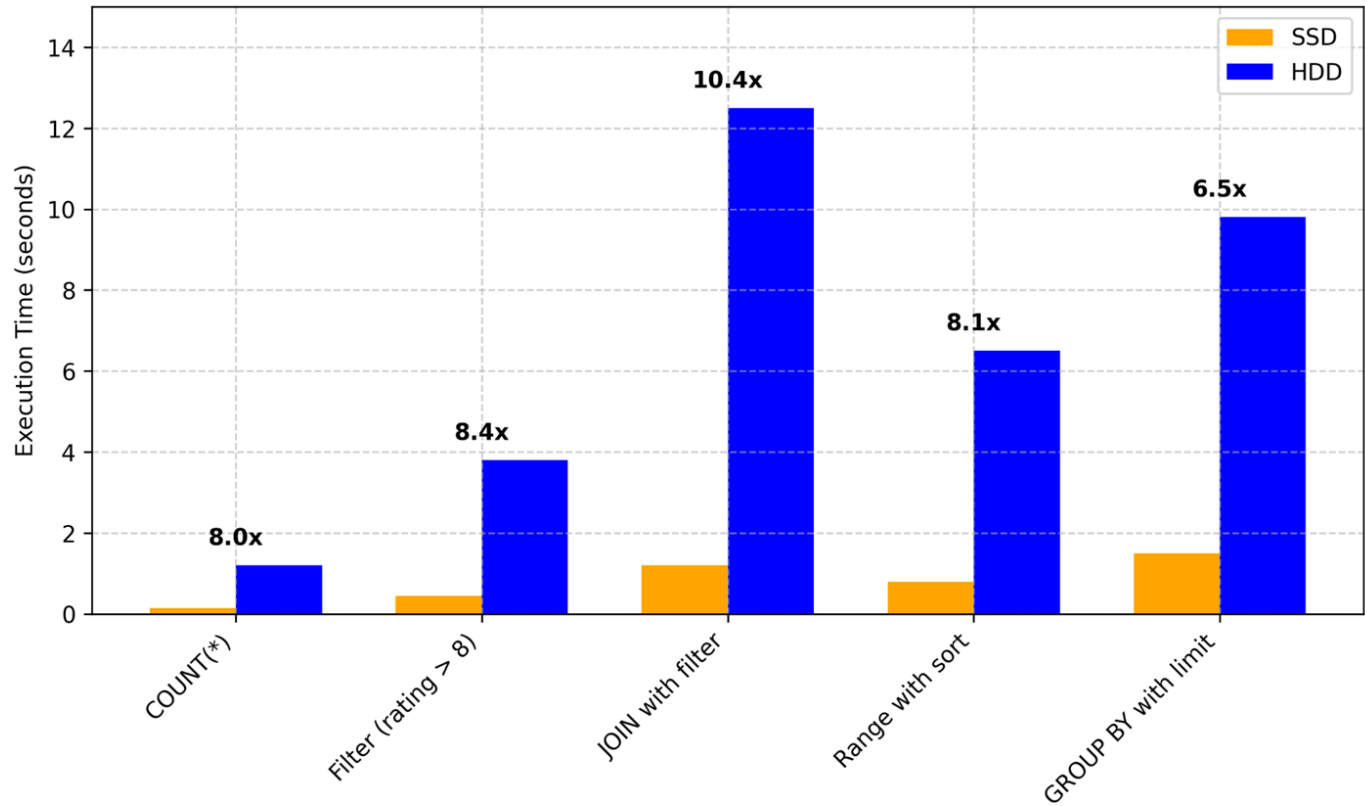
```
docker run -it --rm \  
    --device-read-bps /dev/sda:100mb \  
    --device-write-bps /dev/sda:100mb \  
    -v "$PWD/data:/app/data" \  
    my-benchmark-image
```

Solid State Drive

```
docker run -it --rm \  
    --device-read-bps /dev/sda:500mb \  
    --device-write-bps /dev/sda:500mb \  
    -v "$PWD/data:/app/data" \  
    my-benchmark-image
```

Query Processing: Performance Evaluation & Analysis

HDD vs SSD Query Performance Comparison



Noticeable Insights

- SSD's are nearly better across all SQL queries.
- 'JOIN with Filter' contained the most significant IO operations resulting in the largest difference in performance
- 'Count' had the smallest difference as it has the least number of IO operations required.

	COUNT(*)	Filter (rating > 8)	JOIN with filter	Range with sort	GROUP BY with limit
Returned rows	8,000,000	120,000	45,000	150,000	10



Limitations

- 1. Access to Representative Hardware
- 1. Platform Limitation: Apple M1 and Disk Control
- 1. Resource Isolation Isn't Perfect

Solutions

- 1. Running Inside a Docker Container
- 1. Kali Linux VM
- 1. Single Database inside the Container

Summary

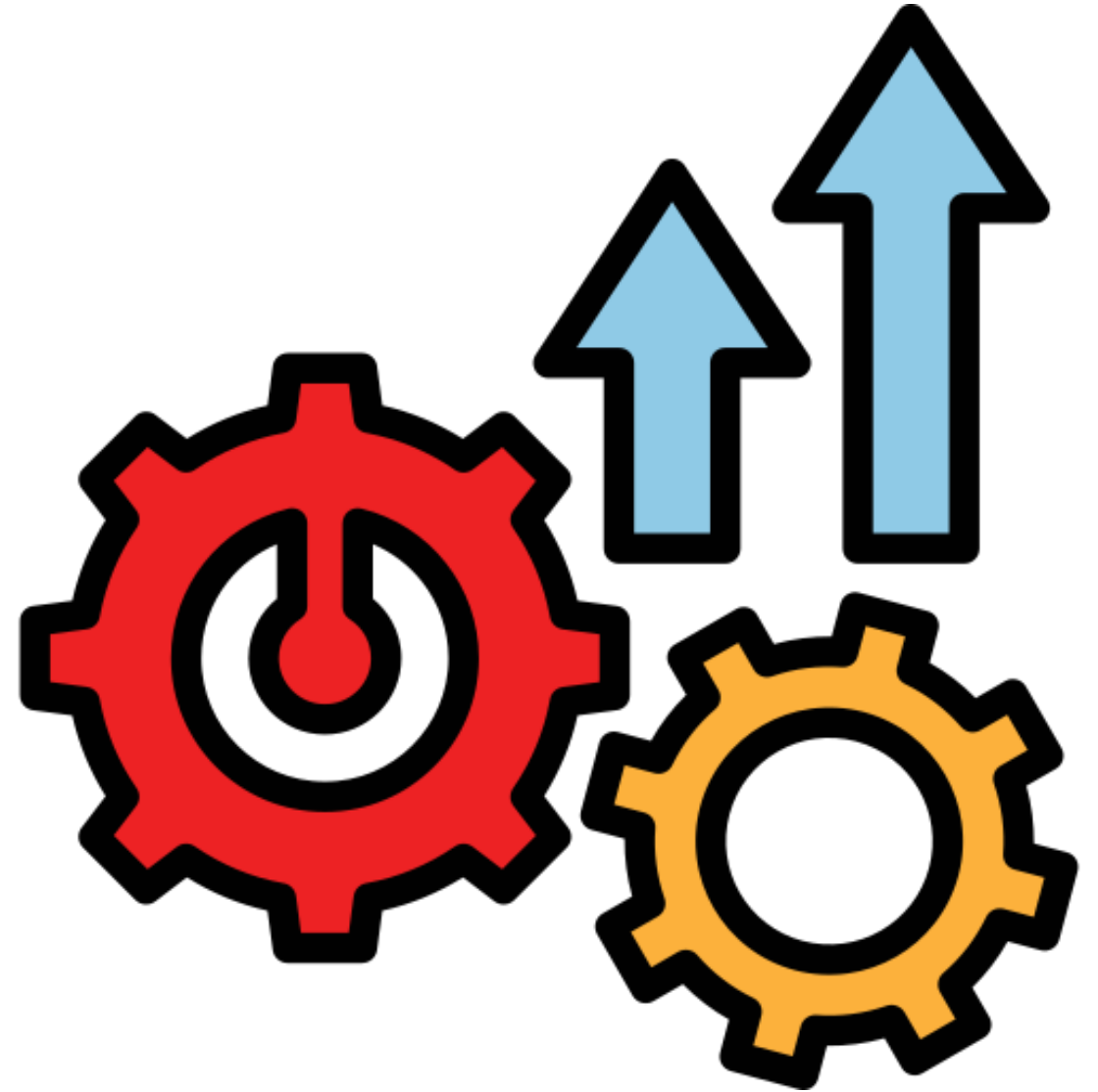
Goals	Checkbox
Performance benchmarking based on different combinations of hardware: <ul style="list-style-type: none">a. Number of CPU Coresb. Amount of RAMc. Number of Threads	
Query processing performance analysis under different disk configurations: <ul style="list-style-type: none">a. SSDb. HDD	

Possible Improvements and Extensions

1. Application Improvement

1. Real-Time Monitoring

1. Streaming vs Batch Processing Comparison



Thank you!