ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

Optimisation de l'accès aux données au CERN et dans la Grille de calcul mondiale pour le LHC (WLCG)

Olga Chuchuk

Centre Inria d'Université Côte d'Azur - équipe NEO

Présentée en vue de l'obtention du grade de docteur en Informatique d'Université Côte d'Azur

Dirigée par : Giovanni Neglia
Co-dirigée par : Markus Schulz
Soutenue le : 19 février 2024

Devant le jury, composé de :
Frédéric Derue, CNRS/IN2P3, Rapporteur
Stéphane Jezequel, CNRS/IN2P3, Rapporteur
Chadi Barakat, Inria, Examinateur
Eric Cogneras, Université Clermont Auvergne, Examinateur
Giovanni Neglia, Inria, Directeur de thèse
Markus Schulz, CERN, Co-encadrant de thèse

Data access optimisation at CERN and in the Worldwide LHC Computing Grid (WLCG)

Résumé

La Grille de calcul mondiale pour le LHC (WLCG) offre une infrastructure informatique distribuée considérable dédiée à la communauté scientifique impliquée dans le Grand Collisionneur de Hadrons (LHC) du CERN. Avec un stockage total d'environ un exaoctet, le WLCG répond aux besoins de traitement et de stockage des données de milliers de scientifiques internationaux. À mesure que la phase du High-Luminosity LHC (HL-LHC) approche, le volume de données à analyser augmentera considérablement, dépassant les gains attendus grâce à l'avancement de la technologie de stockage. Par conséquent, de nouvelles approches pour un accès et une gestion efficaces des données, telles que les caches, deviennent essentielles. Cette thèse se plonge dans une exploration exhaustive de l'accès au stockage au sein du WLCG, dans le but d'améliorer le débit scientifique global tout en limitant les coûts. Au cœur de cette recherche se trouve l'analyse des journaux d'accès aux fichiers réels provenant du système de surveillance du WLCG, mettant en évidence les véritables schémas d'utilisation.

Dans un contexte scientifique, la mise en cache a des implications profondes. Contrairement à des applications plus commerciales telles que la diffusion de vidéos, les caches de données scientifiques traitent des tailles de fichiers variables, allant de quelques octets à plusieurs téraoctets. De plus, les associations logiques inhérentes entre les fichiers influencent considérablement les schémas d'accès des utilisateurs. La recherche traditionnelle sur la mise en cache s'est principalement concentrée sur des tailles de fichiers uniformes et des modèles de référence indépendants. Au contraire, les charges de travail scientifiques rencontrent des variations de taille de fichier, et les interconnexions entre les fichiers logiques influencent de manière significative les schémas d'accès des utilisateurs.

Mes investigations montrent comment l'organisation hiérarchique des données du LHC, en particulier leur compartimentation en “datasets”, influence les schémas de demande. Reconnaissant cette opportunité, j'introduis des algorithmes de mise en cache innovants qui mettent l'accent sur la connaissance spécifique des datasets et je compare leur efficacité avec les stratégies traditionnelles axées sur les fichiers. De plus, mes découvertes mettent en évidence le phénomène des “hits retardés” déclenché par une connectivité limitée entre les sites de calcul et de stockage, mettant en lumière ses répercussions potentielles sur l’efficacité de la mise en cache.

Reconnaissant le défi de longue date que représente la prédiction de la Popularité des Données dans la communauté de la Physique des Hautes Énergies (PHE), en particulier avec la problématique de stockage à l’approche de l’ère du HL-LHC, ma recherche intègre des outils de Machine Learning (ML). Plus précisément, j’utilise l’algorithme Random Forest, connu pour sa pertinence dans le traitement des Big Data. En utilisant le ML pour prédire les futurs schémas de réutilisation des fichiers, je présente une méthode en deux étapes pour informer les politiques d’éviction de cache. Cette stratégie combine la puissance de l’analyse prédictive et des algorithmes établis d’éviction de cache, créant ainsi un système de mise en cache plus résilient pour le WLCG.

En conclusion, cette recherche souligne l’importance de services de stockage robustes, suggérant une orientation vers des caches sans état pour les petits sites afin d’alléger les exigences complexes de gestion de stockage et d’ouvrir la voie à un niveau supplémentaire dans la hiérarchie de stockage. À travers cette thèse, je vise à naviguer à travers les défis et les complexités du stockage et de la récupération de données, élaborant des méthodes plus efficaces qui résonnent avec les besoins évolutifs du WLCG et de sa communauté mondiale.

Mots clés. Algorithmes de mise en cache, WLCG, Popularité des données, Forêt d’arbres décisionnels, Infrastructure informatique distribuée, Big Data

Abstract

The Worldwide LHC Computing Grid (WLCG) offers an extensive distributed computing infrastructure dedicated to the scientific community involved with CERN's Large Hadron Collider (LHC). With storage that totals roughly an exabyte, the WLCG addresses the data processing and storage requirements of thousands of international scientists. As the High-Luminosity LHC phase approaches, the volume of data to be analysed will increase steeply, outpacing the expected gain through the advancement of storage technology. Therefore, new approaches to effective data access and management, such as caches, become essential. This thesis delves into a comprehensive exploration of storage access within the WLCG, aiming to enhance the aggregate science throughput while limiting the cost. Central to this research is the analysis of real file access logs sourced from the WLCG monitoring system, highlighting genuine usage patterns.

In a scientific setting, caching has profound implications. Unlike more commercial applications such as video streaming, scientific data caches deal with varying file sizes—from a mere few bytes to multiple terabytes. Moreover, the inherent logical associations between files considerably influence user access patterns. Traditional caching research has predominantly revolved around uniform file sizes and independent reference models. Contrarily, scientific workloads encounter variances in file sizes, and logical file interconnections significantly influence user access patterns.

My investigations show how LHC's hierarchical data organisation, particularly its compartmentalization into datasets, impacts request patterns. Recognising the opportunity, I introduce innovative caching policies that emphasize dataset-specific knowledge, and compare their effectiveness with traditional file-centric strategies. Furthermore, my findings underscore the "delayed hits" phenomenon triggered by limited connectivity between computing and storage locales, shedding light on its potential repercussions for caching efficiency.

Acknowledging the long-standing challenge of predicting Data Popularity in the High Energy Physics (HEP) community, especially with the upcoming HL-LHC era's storage conundrums, my research integrates Machine Learning (ML) tools. Specifically, I employ the Random Forest algorithm, known for its suitability with Big Data. By harnessing ML to predict future file reuse patterns, I present a dual-stage method to inform cache eviction policies. This strategy combines the power of predictive analytics and established cache eviction algorithms, thereby devising a more resilient caching system for the WLCG.

In conclusion, this research underscores the significance of robust storage services, suggesting a direction towards stateless caches for smaller sites to alleviate complex storage management requirements and open the path to an additional level in the storage hierarchy. Through this thesis, I aim to navigate the challenges and complexities of data storage and retrieval, crafting more efficient methods that resonate with the evolving needs of the WLCG and its global community.

Key words. Caching algorithms, WLCG, Data popularity, Random Forest, Distributed Computing Infrastructure, Big Data

Contents

Résumé [Français] i
Abstract iii
Contents v
List of Figures ix
List of Tables xiii

1 Introduction **1**
1.1 LHC at CERN and the ATLAS experiment 1
1.1.1 ATLAS Data Processing Workflow 2
1.2 Worldwide LHC Computing Grid (WLCG) 6
1.2.1 Architecture 6
1.2.2 Functional Tasks 8
1.2.3 Main Use Cases 10
1.2.4 Hierarchical Organisation of Data into Datasets 11
1.2.5 ATLAS Data Placement and Processing Policies 11
1.2.6 EOS 12
1.3 Data Management Challenges in LHC Experiments 13
1.3.1 Data Lakes 14
1.4 Introduction to Caching 16
1.4.1 Types of Caching 16
1.4.2 Existing Caching Algorithms 17
1.4.3 Different Scenarios of Caching 19
1.4.4 Evaluating Caching Efficiency 22

2 WLCG Workload Characteristics **25**
2.1 Sources of Log Information in the WLCG 26
2.1.1 EOS Report Logs 26
2.1.2 ATLAS Rucio Logs/Dumps 27
2.1.3 Data Sources Consistency 30
2.2 WLCG Logs Processing Workflow 31
2.2.1 Data Collection and Parsing 31

2.2.2 Data Filtering . . . 31
2.2.3 Data Grouping . . . 32
2.2.4 Derivation of the Operation Types . . . 33
2.2.5 Derivation of File-Specific Metrics . . . 33
2.2.6 Data Cleaning and Data Immutability Assumption . . . 35
2.3 General Access Patterns . . . 36
2.3.1 Distribution of File Types (Based on Creation/Deletion time) . . . 37
2.3.2 File Sizes Distribution . . . 39
2.3.3 File Accesses Time Distribution (Temporal Locality) . . . 39
2.3.4 Analysis of Content Popularity and Zipf's Law in Caching Performance . . . 43
2.4 Analysis Files Access Patterns . . . 45
2.4.1 Workload Comparisons: Analysis vs. Total Workloads . . . 45
2.4.2 Comparing File Sizes: Analysis Files vs. All Files . . . 45
2.4.3 Number of File Accesses . . . 46
2.4.4 Correlation Between File Size and Popularity . . . 46
2.4.5 Request Rate . . . 47
2.4.6 Dependency Between the Lifetime and Popularity . . . 47
2.4.7 Popularity of the Files: Ranking-Zipf Plots . . . 48
2.5 Dataset-Based Access Patterns . . . 48
2.6 Conclusions . . . 55
2.7 Log Data Processing Pipeline: A Brief Overview of Implementation Details 56

3 Caching in the Context of the WLCG 57
3.1 Problem Statement . . . 57
3.2 Constructing *MRCs* for Equal-size File Traces . . . 57
3.2.1 *OSA*. Samples . . . 58
3.2.2 *LRU* and *OPT*. Samples . . . 58
3.2.3 *OSA*, *LRU*, and *OPT*. Samples. Comparison . . . 59
3.2.4 *OSA*. Influence of Sampling . . . 61
3.3 Cache Admission: Impact on Algorithmic Performance . . . 61
3.4 *MRCs* for Heterogeneous File Sizes . . . 63
3.4.1 Optimisation of *MRCs* Through Point-Based Construction . . . 63
3.4.2 Lower and Upper bound of the *OPT* algorithm . . . 64
3.4.3 Lower Bound for *BMR* of Reactive Policies . . . 64
3.5 Exploring Enhanced *LRU* Variants: Implementation of *2-LRU* Caching . 68
3.6 Caching Algorithms Taking into Account Specifics of WLCG Workloads . 68
3.7 Performance Comparison . . . 71
3.7.1 Implemented Enhancement Strategies . . . 73
3.8 Limited Connectivity Throughput Study . . . 74

3.8.1 Implementation Specifics . . . 74
3.8.2 Observations on the Queue Length . . . 75
3.8.3 Full-Cache Points and the Calculation of Hit Ratios . . . 77
3.8.4 Discussion of Experimental Results . . . 80
3.8.5 Prefetching Overhead . . . 85
3.9 Bandwidth Exploration . . . 86
3.10 Conclusions . . . 88

4 Machine Learning-based Caching Policies 89
4.1 Description of the Trace: . . . 89
4.2 Performance of the *Belady* algorithm . . . 89
4.3 Architecture of ML solution (2-Stage Approach) . . . 91
4.4 Predicting Future Accesses . . . 92
4.4.1 Search for Simpler Dependencies . . . 92
4.4.2 The Choice of the Predictive Model . . . 94
4.4.3 Training Decision Trees and Random Forests . . . 100
4.4.4 Reformulating the Problem: Watermarks Training . . . 104
4.4.5 Results of the Prediction Models . . . 107
4.4.6 Feature Importance . . . 110
4.5 Integration into the Caching Policies . . . 114
4.5.1 Condensing Model . . . 114
4.5.2 Watermarks . . . 115
4.5.3 Experimental Results and Discussion . . . 117
4.6 Implementation Details and Computational Complexity . . . 117
4.6.1 *LRU* and *LRU Watermarks* . . . 119
4.6.2 Machine Learning Approach . . . 120

5 Conclusions 123
5.1 Future Directions . . . 124

List of Figures

1.1 Schematic of the Data Processing Workflow in the ATLAS Experiment at the LHC. 3
1.2 Illustration of the Worldwide LHC Computing Grid (WLCG) Infrastructure. 7
1.3 Projected Resource Requirements for the ATLAS Experiment from 2020 to 2036. 13
1.4 Flow of Information within a Data Lake Framework. 15

2.1 Classification of File Operations by Size and Byte Metrics. 34
2.2 Definition of *Update*, *Empty* and *Abnormal* Operations. 34
2.3 Data Turnover at CERN T0 Data Center for LHC Detectors. 36
2.4 Comparison of Read Volume and Read Workload in the ATLAS Instance. 37
2.5 Data Read Activities Breakdown at CERN's T0 Data Center for the ATLAS Experiment. 38
2.6 Visual Analysis of File Size Distribution and Volume Contribution at CERN's T0 Data Center, ATLAS Experiment. 40
2.7 Distribution of File Access Frequency. 41
2.8 Time Difference Between Consecutive File Accesses. 41
2.9 Detailed Distribution of Time Intervals Between Consecutive File Requests. 42
2.10 Distribution of File Access Durations. 43
2.11 Zipf-like Distribution Analysis of File Accesses Across LHC Experiments. 44
2.12 File Size Distribution for AOD and DAOD Files. 47
2.13 File Popularity Distribution for Analysis Files. 48
2.14 Correlation Between File Size and Average Number of Read Accesses. . . 49
2.15 Total Number of Read Accesses by File Size. 49
2.16 System Load Correlated with File Size. 50
2.17 Distribution of Byte Request Rate Averaged Per Second. 50
2.18 Dependency Between Lifetime and Number of Accesses. 51

2.19 Comparative Analysis of Zipf-like Distributions for Analysis and Non-Analysis File Accesses Across the ATLAS Experiment. 51
2.20 Dataset Size Distribution. 52
2.21 Distribution of the Number of Files Across Datasets. 52
2.22 Distribution of File Access Frequencies Across Datasets. 53
2.23 Histogram of File Access Proportions Across Datasets. 54
2.24 Variance in File Access Frequencies Within Datasets. 54

3.1 Comparative Analysis of *File Miss Ratio* Across Different Cache Sizes and Eviction Policies for the LHC Detectors. 60
3.2 Impact of Trace Sample Size on *File Miss Ratio* for the *Optimal Static Allocation* (*OSA*) Eviction Policy in the LHC Detectors. 62
3.3 Detailed Comparison of Various Cache Eviction Policies' Performance in Terms of *FMR* and *BMR* Depending on the Cache Size. 72
3.4 Comparative Analysis of FMR and BMR Excluding Cold Misses for Various Cache Eviction Policies. 73
3.5 Queue Size Variation Over Time for *LRU* Cache Policy Under Different Network Throughputs. 78
3.6 Queue Size Variation Over Time for *Dataset LRU* Cache Policy Under Different Network Throughputs. 79
3.7 Hit Rate Dynamics Across Different Cache Sizes for 100 Gbit/s Throughput Using *LRU*. 81
3.8 Analysis of *LRU* Cache Performance Across Various Throughput Levels. . 82
3.9 Analysis of *Dataset LRU* Cache Performance Across Various Throughput Levels. 83
3.10 Comparative Analysis of *BMR* for *LRU* and *Dataset LRU* Under Different Network Throughputs. 84
3.11 Optimal Caching Policy Based on Connectivity Bandwidth and Cache Size. 85
3.12 Prefetching Overhead as a Function of Cache Size. 86
3.13 Distribution of Cumulative Bandwidth Utilisation for File Access Events Over Time. 87

4.1 Comparing the Performance of *LRU*, *Belady* and the Lower Bound of the Optimal Algorithms. 91
4.2 Comparative Analysis of Access Frequency and Access Rate Over Time. . 93
4.3 Overlay Histogram of Normalised Access Times on Log Scale by Access Class. 94

4.4 Correlations between Previous and Next File and Dataset Access Times. . 95
4.5 Decision Tree and Random Forest Schematic Representation. 99
4.6 Regression Analysis: Comparison of Actual vs. Predicted Reuse Distance Distributions. 109
4.7 Regression Analysis: Actual vs. Predicted Distribution of Reuse Distances. 109
4.8 Regression Model Error Distribution by Reuse Distance. 110
4.9 Predicted Class Distribution in Classification Model. 111
4.10 Error Magnitude by Prediction Class in Classification Model. 112
4.11 Feature Importance for Regression Model. 113
4.12 Feature Importance for Classification Model. 114
4.13 Comparing the Basic *LRU* and *Belady* Cache Eviction Policies with Their Watermark-Based Counterparts (Wmk). 116
4.14 Comparing *LRU*, *Belady-Watermarks* Models, with the ML-Based Solutions. 118
4.15 Performance Evaluation Plot of an ML Regressor with Watermark Thresholds. 119

List of Tables

2.1 Description of Main Non-Deletion Log Metrics of EOS Report Logs. . . . 27
2.2 Description of Main FST Deletion Log Metrics of EOS Report Logs. . . . 28
2.3 Description of Main MGM Deletion Log Metrics of EOS Report Logs. . . 29
2.4 Comparison of Files and Number of Accesses in Rucio and EOS Report Logs. 30
2.5 Fraction of *Other* Operations at CERN T0 Data Center for LHC Detectors. 35
2.6 Comparison of the Total Access and Analysis-Only Access Workloads. . . 46

4.1 Main Characteristics of the Updated 2022 File Access Trace. 89
4.2 Retention of Records and Composition of Training Dataset. 93
4.3 Comparison of *RMSE* Scores for Different Regression Methods. 97
4.4 Filtering Reduction Rates of the Training Dataset for Random Forest Model. 106
4.5 ML Model Performance Comparison (Random Forest). 108
4.6 ML Model Performance Comparison (Decision Tree). 108
4.7 Regression Model Performance Comparison. 113
4.8 Impact of the Cache Size on the Number of Cache Cleanups, Simulation Time and the Average *RMSE* of the Random Forest Classification Task. . 116

Chapter 1

Introduction

1.1 LHC at CERN and the ATLAS experiment

The European Organisation for Nuclear Research, better known as CERN [1], is a multinational organisation that serves as the epitome of collaborative efforts in scientific research. CERN is globally renowned for its Large Hadron Collider (LHC) [2], the most powerful particle accelerator in existence. Situated underground near Geneva, the LHC consists of a 27-kilometre ring equipped with superconducting magnets. The primary objective of the LHC is to recreate conditions that existed just moments after the Big Bang, thereby providing an environment for discovering new particles and offering insights into the fundamental forces and constituents of the universe.

The LHC is used by four main detectors, each with a focus on specific research goals, ATLAS and CMS are covering similar topic employing different technologies for calorimetry and the measurement of muons :

- **ATLAS (A Toroidal LHC ApparatuS):** As one of the two general-purpose detectors, ATLAS is designed to investigate a wide range of physics, including the search for phenomena beyond the Standard Model, such as supersymmetry. ATLAS played a significant role in the discovery of the Higgs boson in 2012 [3].

- **CMS (Compact Muon Solenoid):** Like ATLAS, CMS is the second general-purpose detector. CMS also participated in the discovery of the Higgs boson [4].

- **ALICE (A Large Ion Collider Experiment):** This detector is specialised for studying the quark-gluon plasma, a state of matter that existed shortly after the Big Bang. ALICE helps to understand the strong force, one of the four fundamental forces in physics [5].

- **LHCb (LHC beauty):** The LHCb experiment aims to understand why we live in a universe made predominantly of matter. By studying the decay of particles containing beauty and charm quarks, LHCb provides crucial information on the matter-antimatter imbalance [6].

Each detector generates massive volumes of physics data, necessitating a highly efficient computational infrastructure for data storage, management, and analysis, a role fulfilled by the Worldwide LHC Computing Grid (WLCG) [7]. WLCG is a remarkable success of collaboration in the field of computational science, underpinning some of the most significant advancements in computing for particle physics.

In this study, the focus is on the processing of ATLAS detector data. The reason for this lies in the complexity and scale of computing for ATLAS, which is typical for the current generation of HEP experiments at larger accelerators. Additionally, their chosen labelling system efficiently differentiates between data types in the storage system, clearly distinguishing between datasets used for analysis tasks and other forms of data. Analysing the local grid analysis workloads of ATLAS at the CERN Data Denter, the largest center within the WLCG, provides valuable insights into the patterns and structure of data management and processing in the high-energy physics community.

1.1.1 ATLAS Data Processing Workflow

ATLAS, the largest detector at the LHC in terms of physical dimensions and volume, is supported by a global team of over 5,000 members, including physicists, engineers, technicians, and support staff. This collaborative effort, essential for both the construction and operation of the detector, exemplifies international scientific cooperation. The computing and storage needs of ATLAS are distributed among the WLCG sites.

Two Branches of Analysis for Physics ATLAS, like the other LHC experiments, follows two branches of data processing (refer to Figure 1.1). The first begins with data recorded by the ATLAS detector during the operation of LHC. The data stream passes through several steps of selection and filtering with the High Level Trigger (HLT), being the first already implemented in software, deciding on the selection of physics events to keep, leading to initial data filtering and a significant reduction in data rate and so in volume. The collision rate is 40 MHz. After HLT processing, only a fraction of the data is retained for offline analysis, about 1 kHz. The size of each event is typically 1.6 MB [8, 9].

Following this, the raw data (RAW) undergoes so-called reconstruction, which includes calibration, alignment, object identification (particles, clusters, and particle jets), particle track reconstruction, and more. Scientists utilise the reconstructed data for their analyses.

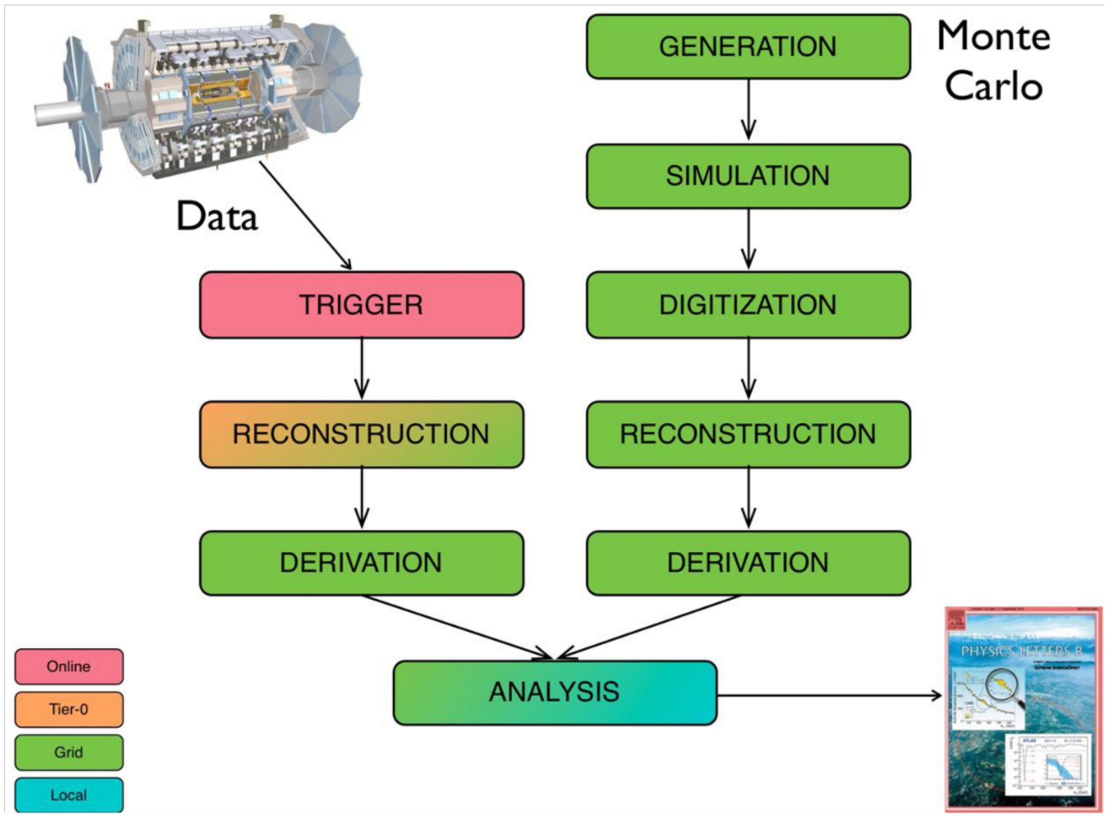


Figure 1.1: Schematic of the data processing workflow in the ATLAS experiment at the LHC. The workflow is divided into two parallel branches: the processing of real collision data (left) and simulated data (right). Real data undergoes initial filtering through the High Level Trigger (HLT) before reconstruction and derivation steps prepare it for final analysis. Simultaneously, simulated data generated via Monte Carlo methods is processed through similar stages, including digitisation and reconstruction, ensuring both data types are suitable for comparative analysis in particle physics research.

In parallel with the processing of real LHC data, another branch handles simulated physics events. This begins with Monte Carlo (MC) generation of collision events based on known physics laws and extensions that are expected to be possible extensions of the current knowledge. This step is called the event generation. The next step of the process then simulates the physical responses of the detector components and the conversion to digital signals, as if these collisions occurred within the detector. This step is called detector simulation, for which the detector response relies either on the GEANT [1] [10] toolkit or a fast simulation utilising parametrised showers in the calorimeter. The resulting MC data closely resembles collision data but also includes so-called "truth" information on the data from the event generators. Subsequently, these data undergo the same processing steps as collision data.

Physicists use both collision and simulated data sources to compare analysis results, confirming or disproving existing theoretical knowledge and making discoveries in particle physics.

Collision Data Flow and Data Model Data from the HLT, in byte-stream format, are permanently stored and archived in the WLCG as RAW data files. These files are further reconstructed into Event Summary Data (ESD) files, which are C++ object representations containing sufficient information for particle identification, track re-fitting, jet calibration, etc. ESD data are in general transient and are kept on disk on demand, with a limited lifespan, and only for a very limited subset of studies. ESD files serve as the basis for deriving subsequent data types suitable for analysis (AOD[2] and DAOD[3]), as well as for control of reconstruction activities and performance monitoring (DESD[4]). As such, ESD files are typically the largest among the three formats: The size of an ESD file for a single event can be in the range of 1-2 MB. AOD files are smaller, usually around 200-500 KB per event. A single event in a DAOD file might be around 50-150 KB, depending on the specifics of the analysis.

[1]GEANT in a software toolkit developed for the simulation of the passage and interaction of particles through matter and fields. It is widely used in high energy physics for designing detectors, studying their performance, and interpreting experimental data. GEANT provides the framework for simulating complex interactions of particles with matter, essential for understanding the experimental outcomes in the ATLAS project.

[2]AOD (Analysis Object Data) files are C++ object representations containing a summary of reconstructed events and sufficient information for common analyses.

[3]DAOD (Derived AOD) - more compact and tailored towards the needs of specific analysis teams.

[4]DESDs (Derived ESD) are reduced ESDs, storing only the information needed for performance evaluation and rate estimates.

Simulated Data Flow and Data Model Data produced by Monte Carlo generators are stored in the EVNT (Event Data) format, a C++ object representation containing "truth" information about particle collisions [8, 9]. Special software [10] simulates the detector's response to these generated events, producing HITS (Hits Data) C++ object representation files that contain simulated energy deposits in active detector volumes and related particle information. HITS files are used for digitising the detector's response and obtaining RDO (Raw Data Object) files, which have the same format as RAW data files but also contain truth information about the events. The subsequent simulation of the High Level Trigger response results in filtered RDO files based on trigger configurations. RDO files are transient and used similarly to RAW LHC data for deriving subsequent ESD, AOD, and DESD files. Each step in this data processing pipeline significantly contributes to the vast amount of data generated in HEP, with final datasets often accumulating to several TB. Specifically, for AODs, the total amounts to 200 petabytes per year for MC simulations and 35 petabytes for the actual detector data.

The first reconstruction of LHC RAW data occurs immediately during or after LHC data-taking, which involves collecting data from particle collisions within the Large Hadron Collider. This provides the basis to ensure data quality and input for the refined calibration and alignment procedures required to exploit the full potential of the data. However, the best precision can be achieved only after processing all data. Therefore, experiments also conduct two types of reprocessing activities:

- **Full RAW $\rightarrow$ AOD Reprocessing:** It initially occurs approximately three months after the initial prompt reconstruction, and subsequently approximately once per year.
- **Fast AOD $\rightarrow$ AOD Reconstruction:** It occurs several times per year in sync with the updates of the calibration software. This process is sufficient if only high-level reconstruction objects (e.g., particle jets) need to be improved.

In cases where lower-level objects, like tracks and clusters, need modification, a full RAW data reprocessing is necessary. Monte Carlo simulated data undergoes reprocessing in parallel with collision data.

ATLAS analysis begins with the AOD files, which have an xAOD format and can be used by both Athena[5] [11] and ROOT[6] [12] to produce final ROOT N-tuples, used in individual analyses.

[5]Athena is the ATLAS software framework managing all steps of the ATLAS production workflows before the final analysis and some physics group analyses.

[6]ROOT is a widely-used data analysis framework in the High Energy Physics community.

AODs are typically too large to run analyses directly. Therefore, they are centrally reduced according to the needs of physics groups, producing DAOD (Derived AOD) files, which have the same format but are much smaller in size through filtering and reducing the information retained.

1.2 Worldwide LHC Computing Grid (WLCG)

Originating from the necessity to manage, analyse, and store massive volumes of data from the LHC, the WLCG has evolved into a complex, distributed computing system, showcasing advanced complexity in its architecture and operations.

1.2.1 Architecture

WLCG follows the Grid paradigm that federates independent resources, allowing user communities to access these resources without being linked to all individual sites as users. This is handled by an abstraction layer called middle ware [13].

The WLCG has a distributed architecture structured in three layers, known as Tiers, which is depicted in the schematic illustration (refer to Figure 1.2) [9].

The Tier 0 (T0) corresponds to the CERN Data Center, which is the largest in the Grid in terms of computational resources and storage capacity [14]. as of 2023, it is equipped with approximately 450,000 processor cores and 10,000 servers, which operate continuously to manage the extensive data generated by the LHC experiments. All data from the LHC detectors passes directly through this central hub. It provides roughly 20% of the overall resources in compute and disk storage; in addition, it maintains a tape-based archive with one copy of all RAW data.

The next layer consists of thirteen Tier 1 (T1) sites, which are large computer centers spread across Europe, Asia, and North America. One of the distinctive features of T1 sites is their direct connection to the CERN Data Center via a high-bandwidth network called the LHC Optical Private Network (LHCOPN) [15]. This network is used to distribute data from T0 to T1 sites. In addition to computational and disk resources, they also provide tape archives that maintain a combined second copy of all RAW data. With their high bandwidth connection, they can handle the data rates required to contribute to the reconstruction effort.

The Tier 2 (T2) sites vary in size from smaller computing and storage facilities, such as universities, to larger at scientific institutes, located around the world. Most T2 sites are connected to the Grid through the LHC Open Network Environment (LHCONE) [16], which enables dynamic allocation of point-to-point virtual networks to further increase

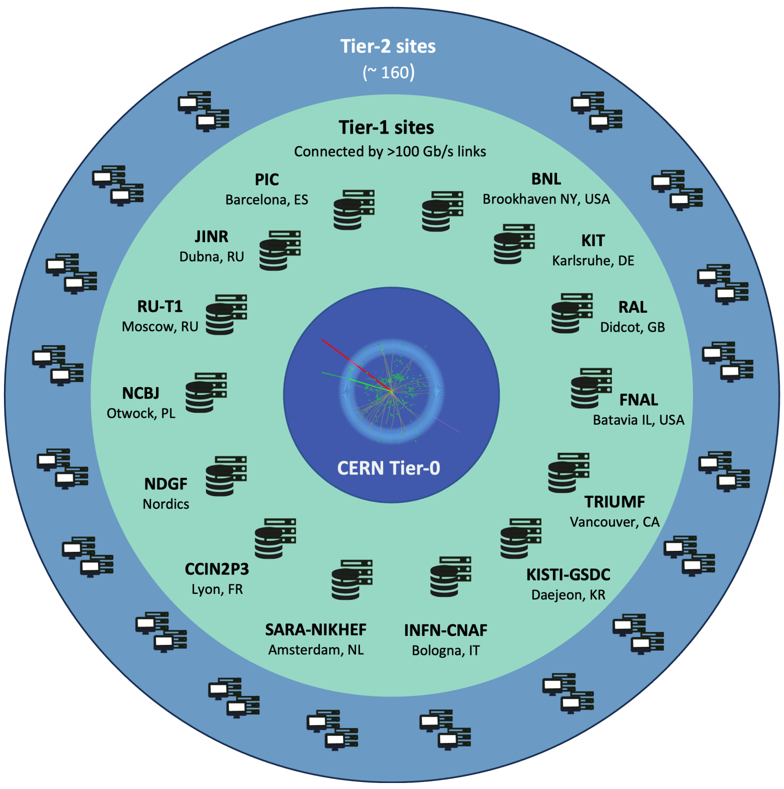


Figure 1.2: Illustration of the Worldwide LHC Computing Grid (WLCG) infrastructure, depicting CERN Tier-0 at the core, surrounded by interconnected Tier-1 sites across various global locations, and the extensive network of approximately 160 Tier-2 sites. Each site is connected through high-speed data links, forming a robust computing ecosystem for LHC data processing and analysis.

network bandwidth. Regular national networks complement this infrastructure, ensuring that all T2 sites maintain connectivity with each other and with T1 sites. In the original design, the connections were expected to be hierarchical, with T2s being assigned to T1s. This was intended to improve the robustness, expecting frequent partial outages of wide area networks. Operational experience proved this as being unnecessary, and the connectivity was converted from a tree to a mesh architecture.

In total, more than 170 active sites are contributing to WLCG. This extensive network collectively contributes approximately 1.4 million computing cores and has a storage capacity of 1.5 exabytes. This setup provides more than 12,000 physicists worldwide with near real-time access to LHC data and the capability to process it efficiently. At the end of the LHC's Long Shutdown 2 (LS2) in early 2022, the global transfer rates between the sites exceeded 260 GB/s.

Additionally, there are local compute resources outside the Grid, ranging from individual laptops or desktops to large analysis facilities. These resources are often referred to as Tier 3 (T3) sites, but are not part of the shared infrastructure and access rights are managed on site-specific terms.

1.2.2 Functional Tasks

The WLCG has to provide the following functionality:

- **Data Archiving:** A long-term commitment to provide reliable storage for the LHC data. Since conducting HEP experiments at accelerators is very costly, experiments are rarely repeated under the same conditions. Therefore, the long-term preservation of data is of great importance. Most of the sites use digital tape for this purpose, as it offers cost-effective, scalable and reliable storage. However, the sites are free to use other storage media for archiving, particularly with the decreasing price gap between tape and disk. The data archiving sites must provide expertise in managing large-scale computing facilities and perform regular validation of data accessibility. Data losses must be minimised and, in the case of one, the data must be recovered from copies at other grid sites. Additionally, these sites should have a good network connection within the WLCG to support the expected data migrations. This task of archiving the physics data is covered by the T0 and the T1 sites.

- **Large-Scale Data Serving:** To provide this functionality, a site must be capable of holding significant volumes of data sets online on disk and serving them over the network to other grid sites. Similar to data archiving, this task requires significant expertise in the management of large-scale storage systems and an adequate network connection with other sites.

- **Compute Facilities:** Depending on the I/O intensity of the computational tasks, the requirements can be divided into two categories:
  - **Reconstruction/Reprocessing:** These tasks are I/O-intensive and, therefore, require that the compute cluster is complemented by large local storage resources, where the data will be populated from an archive facility or other data-serving site. Consequently, these sites also need adequate network bandwidth. Both the storage and network bandwidth must scale together with the size of the compute cluster.
  - **Simulation:** The simulation type of work has relatively little I/O in relation to the required processing needs and, therefore, requires little or no permanent local storage. Sites with sufficient network connectivity to manage remote I/O are suitable for this type of workload. Furthermore, if the output data is not time-critical, these tasks can be run at relatively less well-connected sites if they have a reasonable amount of local storage where the output data can be buffered before being staged to the computing grid for storage and further processing.
- **Infrastructure or Experiment Central Services:** The services supporting the operations of WLCG need servers to run. They include, among others:
  - **Workflow Managers:** Production and Distributed Analysis (PanDA) system used by ATLAS [17], the CMS Remote Analysis Builder (CRAB) [18] and others.
  - **Data Managers:** Rucio [19], which was originally developed to meet the requirements of the high-energy physics experiment ATLAS and is now also used by various other scientific communities; the Distributed Infrastructure with Remote Agent Control (DIRAC) [20], which provides a more generalised workload and data management functionalities, and others.
  - **The VOMS (Virtual Organisation Membership Service):** service for authentication and authorisation [21].
  - **WLCG Information System:** contains current information about the running services [22].
  - **Grid Configuration Database (GOCDB):** stores information about the topology of the Grid [23].

  - **File Transfer Service (FTS)** [24].
  - **Other Grid Services:** VObox [25], a specialised grid service designed to support the needs of a Virtual Organisation (VO) within the grid, CernVM-FS [26], a system to efficiently distribute the experiments' software on WLCG, and others.

The sites hosting them must provide high availability and reliability, as these services are critical for the operation of the infrastructure of the experiments. As of now, most of these services are provided by T0 and T1 sites.

1.2.3 Main Use Cases

Different steps of the data processing vary in terms of computing requirements:

- **Calibration and Alignment:** After passing through the High-Level Trigger, the data undergoes calibration and alignment activities. These tasks are run at the T0 as they require access to the most recent data samples and proximity to the acquisition cluster for immediate feedback. Organised as batch activities, they are labour-intensive and are performed under the control of the core computing teams of the experiments.

- **Reconstruction/Reprocessing:** These activities require access to the RAW data and are therefore organised close to the archives (mostly at T0 and T1, with approximately 10% at T2). Like calibration and alignment, they are organised batch activities run by the computing teams.

- **Stripping, Creation of AODs, Derived Datasets, etc.:** These activities require access to the full processed datasets and are I/O intensive. Thus, they are performed at sites with sufficient storage and access to the archived data (mostly T1s and sometimes the T0).

- **Simulation:** Simulation tasks are compute-intensive but have relatively little I/O. They require compute clusters with enough storage to cache the output while it is being staged elsewhere for longer-term storage and/or further processing. T2 sites, as well as opportunistic resources like clouds and HPC systems, are commonly used for this purpose. The subsequent steps of processing simulated data are analogous to those for collision data and follow the same principles of computing distribution.

- **Group Analysis:** This requires sufficient storage capacity for the input physics data, adequate compute resources, and WAN connectivity to access the input and distribute the output to other sites. This typically occurs at T2 sites.

- **Individual Analysis:** This mostly occurs outside the Grid - at T3 sites. However, T2 sites play the role of data-serving sites in these cases and, therefore, must have reliable disk storage and significant network bandwidth with T3 sites. Some aspects of this work are also done at the T0.

1.2.4 Hierarchical Organisation of Data into Datasets

In the context of data management within the WLCG, the concept of a "file" is largely abstracted; its dimensions are often dictated by transfer duration and the capacity of an operating system to process large files. Simultaneously, a "dataset", which is a collection of files that form a meaningful unit of data needed for physics analysis, represents a broader organisational structure. These datasets were initially divided into files based on storage system requirements and for the convenience of users processing the data.

It's noteworthy that within these datasets, the files, just like the physics events within a file, are independent and have no particular meaningful order. This approach reflects the needs of the WLCG's complex data environment. These files, distributed across different hosts, are predominantly processed in parallel, enhancing efficiency and speed. In certain experiments, overlapping files may be present across multiple datasets.

1.2.5 ATLAS Data Placement and Processing Policies

In managing the voluminous data from the LHC experiments, the ATLAS experiment employs targeted data placement and processing strategies. RAW data, vital for long-term research, are archived permanently at T0 and one T1 site, with additional current-year copies on T1 disks. In contrast, ESD is less frequently archived, with about 5% retained temporarily for processing and assessments. AOD follows a more conservative approach, with one archived copy at Tier 1 and a dynamic replication policy at T1 and T2 sites, contingent on usage.

For simulation data, the policies are equally nuanced. EVNT files are primarily replicated at T2 sites, with distribution as required. High CPU-intensive GEANT simulation outputs (HITS files) are singularly archived at a T1 site.

Derived AODs (DAODs) undergo post-reconstruction processing (skimming, slimming, thinning) to cater to specific analysis needs, leading to a diverse array of smaller but numerous DAOD versions. Notably, these often overlap in events, adding complexity to version management. ATLAS is addressing data management challenges by introducing new, compact data formats like PHYS and PHYSlite [27]. These formats are tailored to reduce data size substantially while retaining crucial analysis information. This innovation aims to standardise data formats across various analyses, enhancing data sharing and efficiency within the scientific community.

DAODs are typically stored in proximity to the research groups that utilise them. This approach aims to facilitate easier access to the data for those specific groups. However, such a localised placement strategy may not always result in an evenly balanced system in terms of data distribution and access.

ATLAS categorises replicas into primary (always available on disk) and secondary (extra copies subject to automatic cleanup). The categorisation is dynamic, based on data access frequency, with T1 datasets becoming secondary after 6 months of inactivity and T2 after a year.

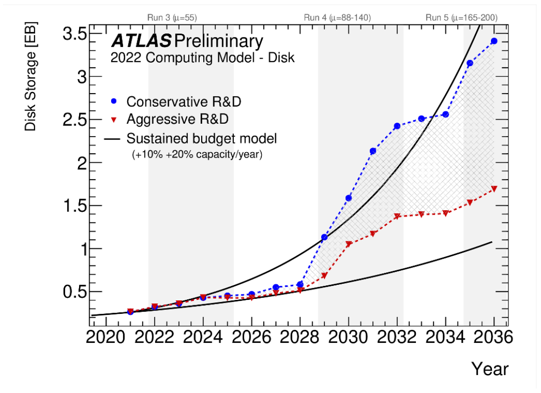
Lastly, space management involves proactive monitoring, with groups manually scheduling deletions to optimise storage efficiency. These comprehensive policies underscore ATLAS's commitment to balancing data preservation with practical storage and accessibility requirements in high-energy physics research.

1.2.6 EOS

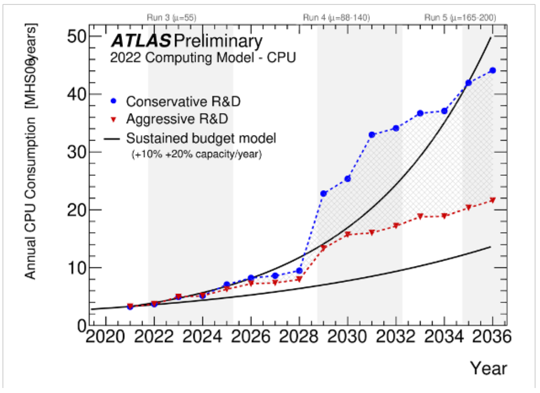
EOS [28] is a large disk storage system that has been developed at CERN since 2010. It serves as multi-purpose storage software and is utilised for storing both physics and user data at CERN and other WLCG sites. EOS is optimised for large-scale installations and provides secure access to data through multiple protocols and supports multi-user access control.

Currently, CERN operates numerous EOS instances, including ones corresponding to specific LHC experiments: LHCb, CMS, ATLAS, and ALICE. Each EOS instance consists of two metadata servers (also known as headnodes or MGM) and up to several hundred disk servers (storage nodes or FST). Each headnode maintains a list of file names along with information about their creation, the number of file copies, file ownership, and other metadata. Storage nodes, on the other hand, house 24–72 disks that store file contents. In total, EOS at CERN manages more than 100,000 disks and houses 8.27 billion files. The total data size currently stored on EOS at CERN exceeds 780 PB.

Due to its primary focus on physics data analysis, the EOS service is characterised by numerous concurrent user accesses, a significant fraction of random data access, and a high file-open rate.



(a) Disk storage requirements



(b) Annual CPU consumption

Figure 1.3: Projected resource requirements for the ATLAS experiment, comparing conservative and aggressive R&D strategies against a sustained budget model, from 2020 to 2036. The sustained budget model is represented by two lines, indicating a 10% (pessimistic) and 20% (optimistic) annual increase in capacity. Each graph also highlights the operational phases of Run 3, Run 4, and Run 5, indicating periods of increased computational and storage demands.

1.3 Data Management Challenges in LHC Experiments

LHC stands as a significant generator of scientific data. Its detectors capture petabytes of data every second from particle collisions. Although a substantial portion of this data is filtered to retain only the most critical for scientific research, the yearly accumulation still amounts to about 30 petabytes. Managing this colossal volume of data poses a considerable challenge, especially with the upcoming launch of the High Luminosity Large Hadron Collider (HL-LHC) runs in 2026.

Presently, the expected available computing resources will be struggling to keep pace with these increasing demands. Despite a steady budget, which allows for some technological improvements, the growth in resources is insufficient to meet the expanding needs of the LHC's research (Figure 1.3).

In light of these challenges, various research and development initiatives are underway to enhance the efficiency of LHC data processing [29,30], including Data Lake model [31, 32], Data Carousel project [33,34], and Data Popularity studies [35,36]. These efforts are not just about improving software performance, but also about identifying more resource-efficient deployment strategies. A crucial aspect of this research, and a focal point of the work presented in this PhD thesis, is the study of caching as a potential solution.

Implementing caching systems at different sites is emerging as a promising strategy to address the anticipated increase in data processing requirements. The benefits of such an approach include faster data transfer times, reduced need for storage space, and fewer data replicas required.

The effectiveness of caching largely depends on the specific data workflows and network availability across sites. An integral part of this thesis involves analysing how storage services are utilised within the WLCG and at CERN's EOS storage system, aiming to effectively evaluate the potential advantages of caching. This study is expected to contribute to the understanding and implementation of caching in high-energy physics data management.

1.3.1 Data Lakes

Amongst various architectural, organisational, and technical shifts, a new data management model is emerging: the Data Lake model [33, 34], a storage service comprising geographically distributed data centers interconnected by a low-latency, high bandwidth network. In this model, caching systems play a critical role from an infrastructural perspective, helping to mitigate latency and improve access to popular data.

The envisioned Data Lake architecture allows for the dynamic movement of data between various Data Lakes. This capability enables the system to store frequently accessed datasets in multiple locations, enhancing data redundancy and availability. This structure is designed to maximise data accessibility and efficiency, ensuring that important datasets are readily available where and when they are needed most.

The architecture integrates different types of centers, each with specialised roles (as depicted in Figure 1.4). Some centers focus on long-term data archiving, using less efficient storage mediums, ensuring data preservation. Others are tailored towards immediate data and computational needs, featuring faster storage options and computational resources. These centers are strategically equipped to handle different aspects of data storage and processing, from high-speed caching to direct network access for data retrieval.

The key aspect of this architecture is the emphasis on cache systems, which play a pivotal role in streamlining data access. This approach not only speeds up data retrieval and processing but also helps in optimising the storage infrastructure, effectively balancing performance with storage needs. This makes the system both efficient and adaptable to various data usage scenarios. Caching is also an attractive option for HEP data because at a given time only a limited fraction of the overall data is actively used.

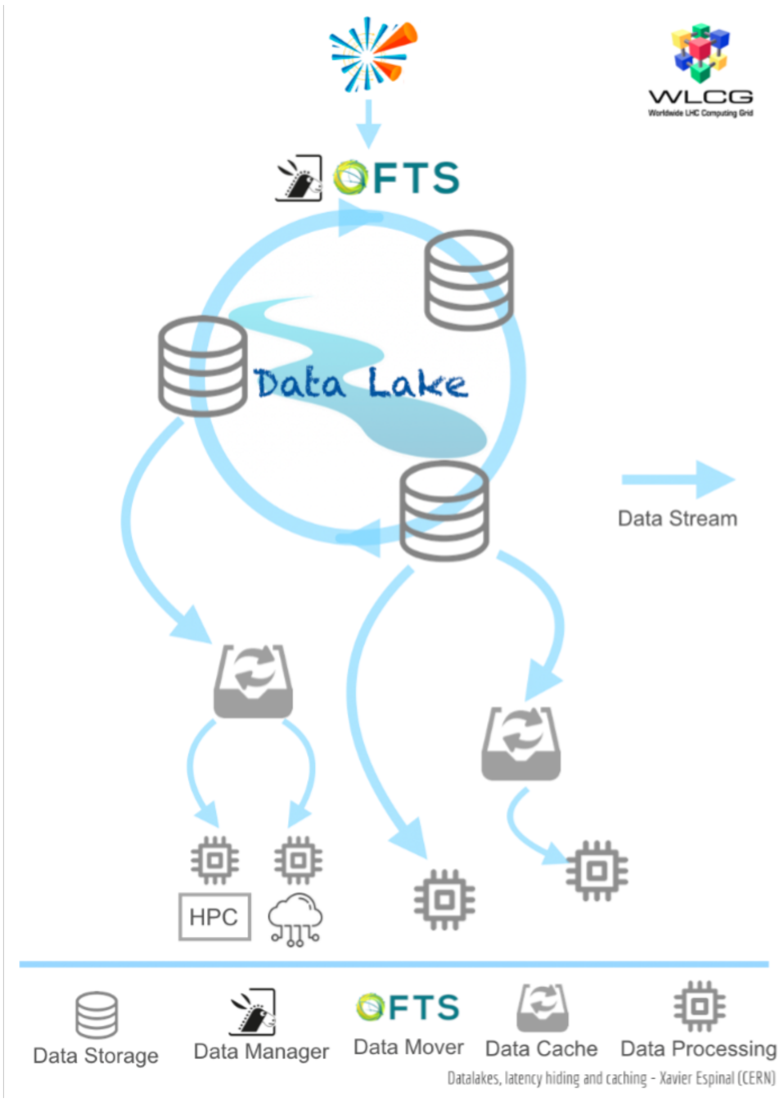


Figure 1.4: Diagram depicting the flow of information within a Data Lake framework, illustrating the interaction between data storage, management, transfer, caching, and processing as implemented in the WLCG infrastructure.

1.4 Introduction to Caching

Before venturing into the specifics of caching within the WLCG framework, it is crucial to understand what caching is in a broader context. At its core, caching involves storing data in a temporary storage area, known as a cache, which allows for faster access to frequently used data. The primary purpose of caching is to reduce access time and improve data retrieval speed by keeping copies of data at points closer to where it is needed. This helps in reducing latency and speeds up data retrieval, improving overall system performance while lowering the used bandwidth on the networks connecting sites. The principle of caching has been applied across a variety of computing domains, from web content delivery to processor operations.

1.4.1 Types of Caching

Based on the application domain and the specific level at which a cache operates within a computing system, various types of caches are distinguished. Some of the common types are:

- **CDN Caches:** Content Delivery Network (CDN) caches are crucial in web content distribution. They store web content at geographically dispersed locations to ensure faster delivery to users irrespective of their location. By caching content like web pages, images, and videos closer to the end-users, CDNs reduce latency and bandwidth usage, thus improving user experience and content accessibility.

- **Web Caches:** Web caching involves storing parts of web pages, such as HTML files and images so that they are quickly available to users on subsequent requests. This type of caching is instrumental in reducing server load, bandwidth requirements, and latency, leading to faster web browsing.

- **Disk Caches:** Disk caching is the process of storing data in a computer's memory (RAM) that is typically stored on a disk. This form of caching accelerates data retrieval processes, as accessing data from memory is significantly faster than disk reads.

- **CPU Caches:** CPU caches are small-sized, faster memory locations within a processor. They store copies of data from frequently accessed main memory locations. The aim is to reduce the time to access memory and to alleviate the bottleneck due to the difference in speed between the CPU and the main memory.

Each type of cache serves a specific purpose and is optimised for the data and access patterns typical to that environment.

1.4.2 Existing Caching Algorithms

The efficacy of the cache system depends on its size and the cache eviction policy, which is an algorithm that determines which files to remove when it reaches full capacity. These algorithms manage how data is stored, retrieved, and evicted from the cache.

To evaluate the performance of a cache system, it is essential to understand the following key terms:

- **Cache Hit**: A cache hit occurs when the requested data is found in the cache memory. It indicates that the data being accessed is already present in the cache, thereby avoiding the need to retrieve it from the slower main memory or external storage. Cache hits are desirable as they result in faster access times and improved system performance.
- **Cache Miss**: A cache miss occurs when the requested data is not found in the cache memory. In this case, the system needs to retrieve the data from the slower main memory or external storage, resulting in longer access times compared to a cache hit. Cache misses are inevitable but can be minimised through efficient caching strategies.
- **Cold Miss**: A cold miss refers to the type of cache miss that occurs when the cache is initially empty. Cold misses typically occur when the system is first initialised and are, therefore, inevitable.

These terms are crucial for measuring the performance of caches, depending on the cache eviction policy.

In this subsection, the most common types of caching algorithms are introduced.

***LRU* and Alike**

The *Least Recently Used* (*LRU*) cache eviction policy, to date, remains highly popular in practical applications due to its straightforward implementation and maintenance, coupled with its capacity to significantly enhance system performance even with a modest cache size, approximately 20-50% of the main storage capacity. This efficacy is documented in various studies [37,38]). *LRU* is particularly effective for workloads exhibiting substantial temporal locality, where requests for the same file are often clustered within short time intervals.

Building upon the *LRU* concept, the *2-LRU* algorithm offers a variant approach. While it shares similarities with *LRU*, *2-LRU* is less stringent in terms of temporal locality. It uniquely focuses on giving priority to the most recently accessed objects, but does so within the subset of recently requested items. This approach is explored in further detail in research [38, 39]. This nuanced difference allows *2-LRU* to manage cache eviction in a manner that can be more suitable for specific types of workloads.

LFU **and** ***OSA***

The *Least Frequently Used* (*LFU*) caching strategy delves into a different dimension of request patterns, focusing on the popularity distribution of individual objects. This approach is well-supported by research [38, 40], which highlights its efficacy. *LFU* is particularly effective in environments where a relatively small subset of objects are more frequently requested than others. By prioritising these items, *LFU* ensures higher cache hit rates for the most popular objects.

In the context of the *Independent Reference Model* (*IRM*)[7], *LFU* demonstrates a close approximation to the *Optimal Static Allocation* (*OSA*) algorithm. *OSA*, an offline algorithm, is designed to retain the most popular objects in the cache, as determined by the observed distribution in the request trace. This alignment with *OSA* allows *LFU* to operate with a high level of efficiency, especially in scenarios where the popularity distribution of requests is skewed towards a limited number of objects.

ARC

The *Adaptive Replacement Cache* policy (*ARC*) [41] stands out as a sophisticated caching algorithm that adeptly considers both the recency and frequency of file requests. By organising files based on recent requests and popularity, *ARC* splits its cache into two distinct logical lists. What sets it apart is its dynamic adaptability: it adjusts the cache space dedicated to each list based on evolving access patterns. Compared to *LRU*, *ARC* offers enhanced performance, being resistant to scans and adept at handling extended periods of low temporal locality, all while retaining the simplicity of *LRU's* implementation.

A salient feature of *ARC* is its intrinsic learning capability. Instead of relying on high-level insights to adapt caching algorithms, *ARC* autonomously discerns access patterns from the ground up. This bottom-up approach underscores the algorithm's ingenuity.

[7]Under this model, the assumption is that requests for different objects are independent and that the probability of requesting an object remains constant over time.

However, adapting *ARC* to scenarios similar to that at the WLCG, with heterogeneous file size distribution, presents challenges. While *ARC* offers a harmonised balance between the *Most Recently Used* (*MRU*) and *Most Frequently Used* (*MFU*) files, no known modifications of *ARC* are tailored explicitly for diverse file sizes. Despite the inherent complexities, the potential for leveraging its foundational principles in heterogeneous contexts remains a promising avenue for exploration.

1.4.3 Different Scenarios of Caching

Caching for Heterogeneous File Sizes

It's imperative to emphasise that the majority of existing literature predominantly examines scenarios with uniform file sizes [37–43]. In contrast, the present research gravitates toward addressing the complexities of heterogeneous file sizes. A common simplification offered in literature is the partitioning of varied file sizes into consistent, standardised chunks. However, this strategy is not universally applicable and can introduce significant computational challenges. The nuances of managing files of disparate sizes require more tailored solutions than merely standardising their dimensions. In this research, files are considered as atomic cacheable units.

LFU, *LRU*, and *2-LRU* can also operate with heterogeneous file sizes with minor changes, but more sophisticated policies in general require a non-straightforward adaptation, e.g., in the case of *ARC*. The existing policies that were specifically conceived to take into account file size: *AdaptSize* [44], *GDSF* [45], *GD-Wheel* [46]), and *DynqLRU* [47].

Caching with Non-Zero Download Delay

The common assumption in caching systems is that file retrieval from remote storage to the local cache occurs quickly enough to precede subsequent requests. This presumption often leads to the disregard of the actual retrieval time, especially under the belief that high throughput ensures efficient file fetching. However, this scenario changes when the throughput-to-latency ratio varies rapidly, making it crucial to consider the impact of delayed hits on overall system performance.

A "delayed hit" is characterised by the file retrieval time exceeding the time until the next request for that file. In simulations of caching algorithms, this situation can mistakenly be counted as a hit since, according to the eviction policy, the file is deemed to be in the cache. This misinterpretation leads to a logical error, particularly when the balance between throughput and request rate is not stable. Therefore, performance evaluation of caching algorithms requires a more nuanced approach beyond traditional methods.

The paper [42] highlights that simply maximising the efficiency of file retrieval does not necessarily equate to minimising latency, especially when dealing with delayed hits. This revelation points to the need for latency-sensitive caching algorithms. The paper introduces two *LRU* variants and discusses methods for their performance evaluation in contexts where delayed hits are significant.

In a similar vein, The authors of [43] investigate modifications to the *LRU* algorithm to incorporate substantial data retrieval times, focusing on Named Data Networking (NDN) architectures. Their approach includes distinct processes for handling request arrivals and content deliveries, aiming to address scenarios where the time to fetch data might exceed the interval between consecutive requests for the same file, resulting in 'delayed hits'.

Caching with Prefetching

The caching policies described previously are fundamentally reactive, updating the cache only after a miss by inserting the requested file. However, leveraging existing correlations in request patterns offers an opportunity for proactive strategies, like prefetching. Prefetching involves forecasting future content requests based on observed patterns and preloading this content into the cache.[8]

In the realm of video streaming, similar to the hierarchical organisation of physics data in the WLCG, data retrieval is both hierarchical and selective. Unlike the complete retrieval of a video file, users often access specific segments or chunks, mirroring how physicists access certain files within a larger dataset. This selective retrieval is akin to the practices in the WLCG.

The key difference, however, lies in the content's nature and utilisation. Video streaming focuses on continuity, ensuring the next segment is preloaded for a smooth experience. This contrasts with the WLCG's approach, where datasets comprise independent files, prioritising specific data retrieval or parallel processing rather than sequential continuity. Despite their shared use of hierarchical and selective retrieval, the objectives and operational nuances of these systems are distinct.

This approach is particularly pertinent to video streaming, as investigated in studies like [48, 49]. Video files, often accessed sequentially, present unique challenges such as handling pauses, fast-forwards, and rewinds.

The study [48] delves into prefetching in the context of sequential data reading in

[8]The terms "caching policies" and "prefetching policies" are often used in literature to differentiate what is referred to here as "reactive caching policies" and "caching policies with prefetching," respectively.

video streams, addressing linear access patterns like playing videos in order, while also accommodating non-linear interactions like pausing or rewinding. This methodology contrasts with the diverse data access patterns at CERN, where, as highlighted in Section 2.5, requests within the same dataset are highly correlated, but file access occurs in an arbitrary order, subject to user and system software preferences.

The concept of dataset-based caching, while highly relevant in the context of CERN's data access patterns, represents a relatively unexplored area within the literature. This approach deviates from the more linear patterns typically associated with video streaming, offering a novel perspective on caching strategies. Therefore, adapting prefetching techniques to accommodate the unique, non-sequential file access patterns at CERN requires a more versatile and complex strategy than those currently employed in standard streaming scenarios.

Machine Learning in Caching

Innovative approaches in cache content management and optimisation have been explored in recent research, focusing on the application of Deep Neural Networks (DNN) and Reinforcement Learning (RL) techniques. The study [50] suggests the pre-training of a DNN to enhance the real-time scheduling of cache content in heterogeneous networks. This concept is further extended by another research [51] which employs a Deep Recurrent Neural Network to predict cache accesses, thereby aiding in effective caching decisions. However, this approach has so far been limited to smaller cache and synthetic datasets, not extending to the larger volumes seen in Data Lakes.

A different angle is presented in the study [52], where predictive models are used to refine the eviction process in fixed-size caches. Another research initiative [53] explores the use of the Gradient Boosting Tree for automating cache management in distributed data clusters. These studies highlight the importance of the environment in cache management, underscoring the need for flexible and autonomous solutions capable of adapting to varying conditions.

To address this need, recent techniques based on RL have been proposed. In [54], the authors use Deep RL for caching popular content across distributed caching entities in the context of Content Delivery Networks. However, this method's online adaptation capability is tested with a limited number of files in a hierarchical caching system. In High Energy Physics, where file sizes vary significantly, this approach might not be directly applicable.

Another aspect of ML in cache management was explored through learning from *Belady* algorithm. The pioneering efforts in this direction were made by [55, 56]. These projects used binary classifiers to predict cache line reuse, either as “cache-friendly” or “cache-averse”, and applied traditional heuristics for eviction order. While effective in overcoming direct learning challenges, these methods were limited in their ability to emulate *Belady* algorithm fully. Contrasting this, recent studies have approached cache replacement as an imitation learning problem, allowing for a more comprehensive policy training that better approximates *Belady* algorithm [57].

Overall, these studies indicate a growing trend towards leveraging ML for more efficient caching strategies in various contexts, including high-energy physics data management and web-based systems. The advancement in this field promises potential improvements in data management and access efficiency, particularly in large-scale computing environments like the WLCG.

Caching for Scientific Workloads

Previous studies on caching for scientific computation often focus on design and deployment of the caching infrastructure [58–60], rather than on selection of well-suited caching algorithms. For example, in [58], the authors simply rely on XCache [61] and its internal implementation of the *LRU* policy. Alternatively, paper [62] explores the usage of a caching technique significantly different from *LRU* for scientific workloads. The authors propose an adaptive caching solution that is only suitable for tasks with high re-execution rates, which are not present in the WLCG.

Some works explore how effectively the WLCG storage is used: at individual grid sites [63], or throughout the whole WLCG in the context of a single LHC experiment [36]. These papers also describe data access patterns potentially relevant for caching, but they do not directly investigate caching strategies.

In this research, another study of the data access patterns in the WLCG is performed, but with a specific focus on those characteristics that directly influence cache performance. Additionally, several new caching policies are proposed, and their behaviour is compared with the existing ones under different scenarios (cache size, network connectivity throughput).

1.4.4 Evaluating Caching Efficiency

The optimal cache eviction policy maximises the number of cache hits compared to the number of cache misses.

The standard metric used to evaluate the cache performance is the hit/miss ratio, but it fails to capture the different costs of different misses when file sizes are heterogeneous.

Therefore, it is useful to distinguish the hit ratio, also called *File Hit Ratio* (*FHR*), and the *Byte Hit Ratio* (*FHR*). They can be defined as follows:

$$FHR = \frac{N_{cache}}{N}, \quad BHR = \frac{V_{cache}}{V}, \tag{1.1}$$

where N is the total number of requests (or the trace length), V is the total volume of the catalogue, i.e., the total size in bytes of all files which have been requested at least once, N_{cache} is the number of files retrieved from the cache (the total number of hits), and V_{cache} is the total number of bytes served by the cache.

Respectively, *File Miss Ratio* (*FMR*) and *Byte Miss Ratio* (*BMR*) are calculated as:

$$FMR = 1 - FHR, \quad BMR = 1 - BHR. \tag{1.2}$$

Under different scenarios, one of these metrics can play a more important role than the other. For example, *FMR* is more relevant if the objective is to minimise the user delay and the retrieval time under a miss is almost constant (latency dominates the retrieval time). At the same time, *BMR* is more important for assessing the data volume transfer between the sites. From the definition, when the files have the same size, *FMR* and *BMR* coincide.

Lower bound for *FMR* of reactive policies. In case of homogeneous file sizes, *Belady* offline algorithm [64] achieves optimal *FMR* amongst reactive policies [37]. At each step, this algorithm evicts the file that will be requested the furthest in the future. Since it can only be calculated post-factum, practical implementation is not possible. However, in studies similar to the one presented in this work, it can serve as a lower bound for the performance of practical reactive policies.

In the case of heterogeneous file sizes, the reactive caching policy that minimises *FMR* (resp. *BMR*) is denoted as *OPT* (resp. *OPT.Bytes*). Both minimisation problems are NP-hard [65], which means that in practice, finding the exact optimal policies is not feasible.

A simple lower bound for both *FMR* and *BMR* of reactive policies can be computed by simulating an infinite size cache [66]. This approach only quantifies cold compulsory misses, which inevitably occur when a file is requested for the first time. Instead, paper [67] proposes several algorithms to calculate lower and upper bounds for *FMR* of *OPT*. The *Flow-based Offline Optimal* (*FOO*) lower and upper bounds presented by the authors are very accurate but computationally expensive; the *Practical FOO* lower and upper bounds (*PFOO-L* and *PFOO-U*) work for hundreds of millions of requests, while still providing tight upper and lower bounds for *OPT* performance. Paper [68] presents yet another lower bound for *FMR* of non-anticipative policies, but only under some statistical assumptions on the request process. In this work, the *PFOO-L* lower bound is used and extended it to be able to compute a lower bound for *OPT.Bytes*' *BMR*.

Chapter 2

WLCG Workload Characteristics

Before delving into the principles of caching in the context of the WLCG, one first needs to better understand the specifics of the user accesses in this environment. In order to evaluate access and usage of storage, this work uses data access and popularity studies for the analysis workflows executed in the EOS node supporting CERN activities, based on local monitoring data spanning several months. In this chapter, the focus is on a three-month time frame: 01/2020 - 03/2020, and limit the studies to the analysis user accesses present at the CERN Data Center - the WLCG T0 site. This study mostly considers the ATLAS experiment, sometimes along with other large experiments, in order to provide a meaningful comparison.

As already mentioned, the T0 storage is supported by EOS, ATLAS, as well as the three other large experiments, each have their dedicated EOS instance. The nuances of data storage within the EOS infrastructure will be explored in greater depth.

From the storage point of view, each physics file has 3 main types of file events during the lifespan: **creation**, **accesses** (if any) and **deletion**. Due to the specifics of the analysis, the physics files are rarely modified, which is supported by the analysis conducted in Subsection 2.2.6.

Creation is the moment when a file is generated in the system. There can be different use-cases: either it's a new file that came directly from the experiment, or it is a new file that was generated by the physics users as a part of their analysis, or it's a copy of already existing file in the system generated for the service reliability or performance improvements.

The file **accesses**, which could also be seen as file reads due to the almost absent file modifications, play a crucial role in the question of optimising the WLCG storage, and in this study, in particular. The number of accesses during the file existence might vary from zero accesses to a very large number, especially for some files which are often constantly used by the end-users to run their analysis.

File **deletion** time is not solely dependent on the end-user decisions but is also influenced by the deletion policies of the entire experiments. Deletions usually occur in large chunks, especially when storage space is nearing its limit. At the same time, a common case for the EOS storage system is that most of the files are created and are not meant to be deleted according to the experiment data storage workflow specifics – this usually happens for the data files coming directly from the experiment, so for the RAW files. Great care is taken to ensure that these files are securely stored. The policy at LHC is that as a minimum one copy is kept at CERN, in the tape system, and one copy is stored at one of the T1 centers.

2.1 Sources of Log Information in the WLCG

The WLCG have been monitored, for more than a decade, with in-house central solutions gathering and storing in the CERN storage facilities a large amount of metrics and logging information. The monitoring system has been fully developed and supported by the software teams at CERN and the experiments and also make use of CERN cloud resources [69]. The monitoring infrastructure covers the whole workflow of the monitoring data: from collecting and validating metrics and logs to making them available for dashboards, reports and alarms. In the research, two types of logs covering WLCG are used: EOS report logs and Rucio Dumps.

2.1.1 EOS Report Logs

The EOS headnodes generate report log files every day. These log files have a highly structured format and are stored as .eosreport files on the headnodes.

Each EOS report logs file is stored as an archived text file, where each line represents a separate log record, which in turn corresponds to a single file event. Each line contains a sequence of key-value pairs, representing a set of metrics about the concerned file and the concerned event, and encoded in the following way:

```
key1=val1&key2=val2&...&keyN=valN.
```

There are 3 types of EOS Report Log records: a record of the first type is generated each time a file was open or created, the two other types correspond to file deletions from the disk and from the namespace.

Each non-deletion record contains more than sixty metrics. The explanation of the main ones is in Table 2.1:

Term	**Description**
`log`	log record identifier
`path`	logical path to the file
`td`	trace identifier in the format: `<user_name>.<process_id>:fd@<origin_host>.[<domain>.]`
`fid`	file identifier
`fsid`	filesystem identifier
`ruid`	identifier of the user
`ots, cts`	opening and closing of the file timestamps
`otms, ctms`	opening and closing file timestamps in milliseconds
`rb, wb`	bytes read and written during the operation
`osize`	the file size at the opening
`csize`	the file size at the closing

Table 2.1: Description of main non-deletion log metrics of EOS Report Logs.

As mentioned earlier, the process of file deletion is not atomic and usually constitutes of two main steps, which correspond to 2 different subtypes of deletion log formats:

- A version of the file is deleted from a local disk (FST deletion). In this case, other replicas of this file can still exist in the system. Regardless of the number of replicas left, this type of deletion does erase information about this file on the headnode. The description of the format is given in Table 2.2.

- Deletion from the headnode (MGM deletion). This happens when the file is deleted from the system altogether. Usually, this deletion is propagated to the FST where the local versions of the files are deleted. The description of the format is given in Table 2.3.

In this research, the MGM deletion is considered to represent the user workflow, as they only occur when someone explicitly deletes the file. Disk deletions, on the other hand, could happen due to user-unrelated internal system processes, such as balancing.

2.1.2 ATLAS Rucio Logs/Dumps

Incorporating the data management and workflow management logs into internal storage system logs proved to be a valuable endeavour, as it allowed for the correlation of these logs with ATLAS datasets through filenames.

DDM Rucio [70] represents an open-source framework that empowers both scientific collaborations and individual users to efficiently organize, manage, and access data on a global scale. This framework facilitates the distribution of data across geographically

Term	**Description**
`log`	log record identifier
`host`	FST host name
`fid`	file identifier
`fsid`	filesystem id where the file is deleted
`del_ts`	deletion timestamp
`del_tns`	deletion timestamp in nanoseconds
`dc_ts`	timestamp of the last change in the metadata (e.g., change of the owner, change of the rights)
`dc_tns`	timestamp of the last change in the metadata in nanoseconds
`dm_ts`	timestamp of the last modification of the content (e.g., creation, update)
`dm_tns`	timestamp of the last modification of the content in nanoseconds
`da_ts`	the last local disk access timestamp (not necessarily only user accesses)
`da_tns`	the last local disk access timestamp in nanoseconds
`dsize`	the file size before deletion
`sec.app`	“deletion”

Table 2.2: Description of main FST deletion log metrics of EOS Report Logs.

dispersed data centers. Originally conceived to meet the specific needs of the ATLAS experiment, Rucio offers functionalities such as the aggregation of files into datasets and the management of data distribution and replication within the grid. Last, but not least it provided a grid enabled file catalogue that is used to locate and track files and data sets.

Rucio Traces The data access operations on WLCG, involving WMS (Web Map Service) or DDM (Data Distribution Management) system, are monitored by the Tracer system. This system creates a trace dictionary for each file, recording essential details like filenames, scopes, related datasets, access points, types of access (like Analysis or Production uploads), user information, file specifics, and timing data. These traces are then sent from the Rucio server to a central ActiveMQ broker. From here, different consumers process this data, with the Kronos daemon in Rucio being especially important. Kronos assimilates these traces and updates key parameters in the Rucio catalogue, such as the last access timestamp for files and datasets. This information is crucial for determining the priority of replicas and deciding when to delete them.

Rucio API The Rucio API serves as a comprehensive source of metadata pertaining to dataset attributes, including project information, run numbers, succinct descriptions of physics processes, production tags, and data formats. Furthermore, it offers insights into dataset replicas and their placements. An essential popularity metric provided by this

Term	**Description**
`log`	log record identifier
`fid`	file identifier
`host`	MGM host name
`del_ts`	deletion timestamp
`del_tns`	deletion timestamp in nanoseconds
`dc_ts`	timestamp of the last change in the metadata (e.g., change of the owner, change of the rights)
`dc_tns`	timestamp of the last change in the metadata in nanoseconds
`dm_ts`	timestamp of the last modification of the content (e.g., creation, update)
`dm_tns`	timestamp of the last modification of the content in nanoseconds
`dsize`	file size before deletion
`sec.app`	"rm" or "recycle"

Table 2.3: Description of main MGM deletion log metrics of EOS Report Logs.

API is the `access_cnt`, which quantifies the number of accesses.

In the ATLAS experiment, a unique identification system ensures that the combinations of `scope` and `filename`, as well as `scope` and `dataset name`, are distinct. Leveraging this uniqueness, it was possible to create a mapping table derived from the Rucio dumps. This table links `scope + filename` (`sc + fname`) to `scope + dataset name` (`sc + dname`). This mapping facilitates the enrichment of data gathered from EOS Report Logs with additional dataset information. The integration of this mapping table with the EOS logs allows for a more comprehensive understanding of data access and usage patterns, aiding in better data management and optimisation of resources within the ATLAS experiment framework.

During the research process, a significant challenge was the lack of scope information in the EOS Report Logs, complicating the task of dataset identification. Despite this, it was observed that most filenames and dataset names were unique. A very small percentage (about 0.0015% for filenames and 0.0003% for dataset names) did not maintain this uniqueness. However, these instances were so rare that they had a negligible effect on the overall dataset identification process.

An intriguing issue encountered during the research was the absence of certain files in the Rucio logs. Upon investigation, it was found that many of these missing files had a `.rucio.upload` extension, which was consistently not recorded in the Rucio logs. Discussions with the ATLAS Distributed Computing team revealed that these files, marked by the `.rucio.upload` extension, are used to confirm successful uploads and are intentionally deleted soon after creation. Therefore, they can be excluded from the research analysis without impacting the validity of the study.

Metric	Rucio	Rucio Only	Rucio and EOS	EOS Only	EOS
Accesses	890198	8905	881293 (Rucio) 1177969 (EOS)	990269	2168238
Files	655686	7541	648145	372032	1020177

Table 2.4: Comparison of files and number of accesses in Rucio and EOS report logs at the CERN T0 Data Center for the ATLAS experiment on October 1, 2020.

Furthermore, the remaining files not found in the Rucio logs constituted a small volume, less than 2.5%. By filtering the data to include only read accesses, almost all of these files could be traced back in the Rucio system. This further validates the comprehensiveness and accuracy of the research methodology.

2.1.3 Data Sources Consistency

In collaboration with colleagues from ATLAS, a study was undertaken to evaluate the consistency of data popularity metrics derived from various data sources. This assessment is a crucial phase in the research, as it lays the groundwork for integrating and combining data from multiple sources. The objective was to ensure that the measurements of data popularity—how frequently and widely data is accessed or used—remain reliable and consistent across different datasets and systems.

For this study, a specific time frame was chosen: October 1, 2020, from 00:00 UTC to 23:59 UTC. All Rucio traces with access times within this period, specifically for files located at CERN, were selected. Similarly, file access records from EOS logs during the same period, excluding those related to system events, were also chosen for comparison.

The results of this selection, including the number of accesses and the distinct files involved, were compiled into a Table 2.4.

Upon examining the data, it was found that 98.8% of the files that were accessed according to Rucio were also accessed according to EOS. It is important to note that this comparison was made without attempting to match the exact timestamps of access. The small fraction of discrepancies, accounting for about 1.2%, predominantly consisted of traces that were generated prior to the actual file access. These traces, termed "direct access", often did not lead to actual file access due to interruptions such as job crashes. On the other hand, the alignment was less pronounced in the opposite direction: only 63.5% of the files accessed as per EOS records were found in the Rucio traces. This finding was not unexpected, given that a significant number of files in EOSATLAS (ATLAS EOS instance at CERN) are not accessed via PanDA jobs and therefore are not traced by the ATLAS framework.

The overarching conclusion from this study is that Rucio and EOS generally agree on which files are accessed, as long as the files are successfully opened by the job. This shows that Rucio is reliable in tracking file access, which is important for research that combines data from different sources.

2.2 WLCG Logs Processing Workflow

2.2.1 Data Collection and Parsing

The first step in the preparation of data for the analysis – is the collection of log records from the headnodes into one place. For this purpose, there is a daemon running every night to copy the log files over from the headnodes (where they were generated) to the machinery where they will be processed for further analysis. These background jobs create .eosreport files for each day per each EOS instance. As described previously, the log records have a special text format. To make them more convenient for further multiple accesses, this data was parsed and save in a tabular format, which is more suitable for this analysis. At this stage, the record logs are seperated into three types (event, MGM and FST deletions).

2.2.2 Data Filtering

This step is vital for the analysis since it aims to reduce the volume of data as much as possible and, at the same time, maintain the main metrics that could be of interest to the research. Therefore, in these logs, one can track all the events that happened to a file – creation, read, deletion and the possible rare updates. However, it contains not only user-generated events but also system processes. As this research aims at tracking user analysis workflow, those need to be neatly separated.

The subsequent phase involves filtering the columns, focusing on the metrics of interest:

- **fid:** Can be used as a unique file identifier since this field was programmed as an increasing counter. Moreover, all replicas of the same file share a common `fid`.
- **osize, csize:** The difference between these two sizes serves as an indicator of the operation type described in the log record.
- **ots, cts:** Only these two fields are considered for the timestamps of file opening and closing. The timestamps do not include milliseconds (`otms` and `ctms` fields) since an event with a time difference of less than one second is too short to involve a file opening.
- **rb, wb:** It is important to note that during read operations, the value of `rb` may not always match `csize`, as the entire file may not be read or the file may be read multiple times. For creation operations, the value of `wb` can vary; it may be greater than `csize`, indicating that the process rewrote some parts of the file, or it may be smaller. A smaller `wb` could indicate either an update operation that affected only a portion of the file or concurrent writing by another process to the same file (refer to Subsection 2.2.4 on operation classification).
- **td:** A particular combination of its parts can serve as a process identifier, as described in the next subsection.

2.2.3 Data Grouping

Given that a single session may correspond to multiple records in a log file, it was necessary to develop a method to differentiate between sessions and accurately group records associated with the same session. This required the creation of a global session identifier, which was not explicitly maintained in the log files.

Given the structure of the trace identifier `td` field:

`<user_name>.<process_id>:fd@<origin_host>[<domain>.]`, a global session identifier was created using a substring of it, consisting of the `user_name`, `process_id` (local), and `origin_host`.

After this, all records were grouped based on this session identifier and the file id as a key. The remaining metrics were updated in an aggregated manner:

- **osize:** `osize` from the record with the earliest `ots`;
- **csize:** `csize` from the record with the latest `cts`;
- **ots, cts:** the earliest and latest timestamps, respectively; and
- **rb, wb:** sum of all the `rb` and `wb` fields, respectively.

2.2.4 Derivation of the Operation Types

As outlined in the EOS Report Logs description, most storage system activity logs, except for deletions, follow a uniform format. These logs encompass a range of file operations, including reads, writes, updates, and others. However, the original log metrics do not specify the type of operation performed. Identifying the nature of these operations is crucial for understanding the lifecycle of a file.

To address this, the existing metrics—such as the size of the file upon opening and closing, and the volume of bytes read and written— have been utilised to categorise operations into five distinct classes: *Create*, *Read*, *Update*, *Empty*, and *Abnormal*.

The initial fundamental operation identified is the *Creation* of a file. This is characterised by an opening file size of zero, a positive number of bytes written, and a non-zero closing file size. This operation is distinguished from an *Update*, where the file's opening size is greater than zero. Both *Creation* and *Update* operations may involve non-zero read bytes, as the system might re-read portions of the file post-writing for internal verification.

The graphical representation of this heuristic can be found on Figure 2.1.

In the process of log analysis, numerous *Empty* operations were observed, characterised by zero bytes read and written. Furthermore, some operations exhibited a closing file size of zero despite recording some read or written bytes. These instances have been classified as *Abnormal* operations (refer to Figure 2.2 for details).

After this classification stage, each log record in the data corresponds to a singular operation executed on a file.

2.2.5 Derivation of File-Specific Metrics

To effectively track data access patterns, it was essential to acquire file-specific metrics, particularly those indicating the frequency of operations performed on each file. To this end, the daily operations data was utilised to generate daily file tables. These tables

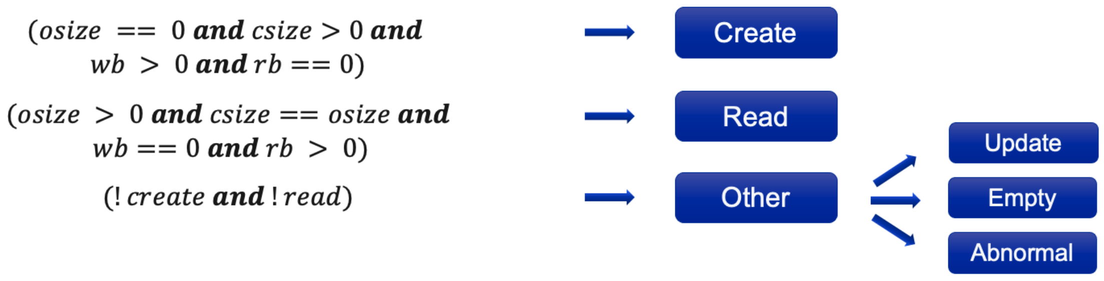

(osize == 0 and csize > 0 and wb > 0 and rb == 0)
(osize > 0 and csize == osize and wb == 0 and rb > 0)
(! create and ! read)
Create
Read
Other
Update
Empty
Abnormal

Figure 2.1: This figure categorises file operations into *Create*, *Read*, and *Other* based on the initial and closing file size, and the volume of bytes read and written. Creation operations are defined by an initial file size of zero and a non-zero number of bytes written. Read operations involve a positive initial file size and non-zero bytes read with no bytes written. Operations not fitting these criteria are labeled as *Other* and further subdivided into *Update*, *Empty*, and *Abnormal* based on additional metrics.

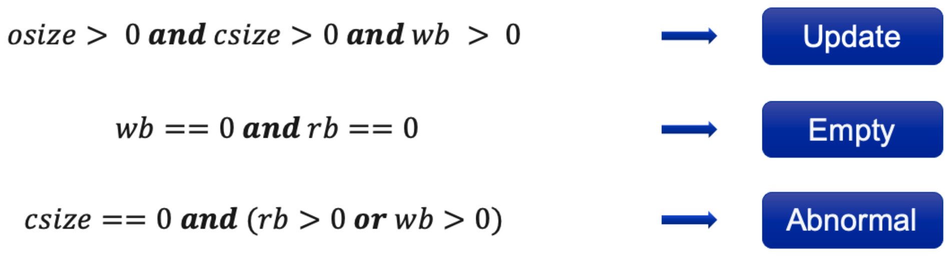

osize > 0 and csize > 0 and wb > 0
wb == 0 and rb == 0
csize == 0 and (rb > 0 or wb > 0)
Update
Empty
Abnormal

Figure 2.2: This figure provides specific criteria for identifying *Update*, *Empty* and *Abnormal* operations within log data. An update is indicated by both non-zero initial and closing file sizes, coupled with bytes written. Empty operations show no bytes read or written. Abnormal operations are defined by a closing file size of zero despite some activity, such as bytes read or written.

include essential information such as file identification, file size, and a count of each operation type, categorised as per the classification outlined in the preceding subsection.

This approach enabled direct engagement with file-level data, facilitating the filtering of information based on file IDs. Most crucially, it allowed for the computation of file-specific metrics, such as the frequency of each type of operation.

2.2.6 Data Cleaning and Data Immutability Assumption

Empty and *Abnormal* operations appear in log records for various reasons. However, it is pertinent to exclude these from this analysis if their occurrence is statistically insignificant. Additionally, an observed anomaly in some files is the occurrence of multiple *reations*. This could be attributed to a file initially being *Created*, then *Updated* (which may involve truncation to zero size), followed by another operation misclassified as a *reation*. For the integrity of this study, files exhibiting this pattern will be excluded, provided they represent a small percentage of the total.

Furthermore, considering the specific workflow on EOS instances for the detectors, most new data, originating directly from experiments, typically undergoes no *Updates*, but is *Read* multiple times instead. The calculation of the proportion of files undergoing *Update* operations validates this observation. Finding this number to be negligible, these files were excluded from this analysis.

The aggregated data on operations excluded from the study are presented in Table 2.5. This includes the percentage of files involved and the corresponding percentage of the total data volume these files represent.

Metric	**LHCb**	**CMS**	**ATLAS**
Other Operations (% of Related Files)	0.06%	0.26%	0.61%
Other Operations (% of Total Volume)	0.89%	0.05%	0.14%

Table 2.5: Fraction of *Other* operations related to the total related files and total volume at CERN T0 Data Center for LHC detectors, Jan-Mar 2020.

In conclusion, it is a reasonable assumption that the data in question is immutable, characterised by a single instance of creation followed by multiple *Read* operations.

Following the data cleansing process outlined in this subsection, this analysis now exclusively focuses on files that exhibit a distinct pattern: precisely one *Creation* operation, accompanied by zero or multiple *Read* operations (hereafter referred to simply as “read” or “read access”), and devoid of *Update*, *Empty*, or *Abnormal* operations. This selection criterion further reinforces the premise that the data under study is immutable.

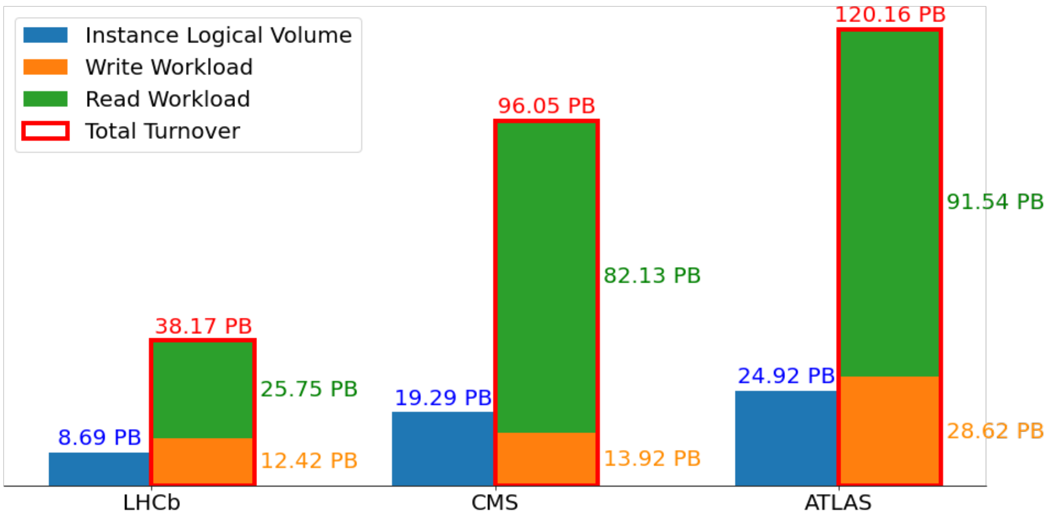


Figure 2.3: Data turnover at CERN T0 Data Center for LHC Detectors. The bar chart illustrates storage use and data turnover for three LHC detectors—cumulative for Jan-Mar 2020. The Instance Logical Volume (blue) shows data stored, Write Workload (orange) captures data written, and Read Workload (green) reflects data read. Total Turnover (red outline) is the aggregate of Write and Read Workloads. The ATLAS experiment, represented by the final bar, demonstrates a notable peak in Total Turnover at 120.16 PB.

2.3 General Access Patterns

As already mentioned, the results of this chapter are based on the log files of three consecutive months (01/01/2020 - 31/03/2020), which is not a data-taking period and the tasks performed at the site mainly consisted of Monte Carlo production jobs and data analysis.

Figure 2.3 gives an overview of the read/write processes happening at the EOS instances during the considered three months and shows how actively the provided disk volume was used by ATLAS and two other LHC experiments. The plot indicates that the access patterns differ from one experiment to another.

As a benchmark and a reference for the experiment's size, the data has been extracted from the EOS Control Tower (Grafana) [71] on each instance size. The total volume is slightly changing over time, but when considering a several months period, an average approximation is sufficient to give a broad idea of the instance's size.

ATLAS, in comparison to the other experiments, has the biggest EOS instance volume and had the most intense workload. Its total turnover (the sum of all the bytes read and written) is over 480% of the instance volume at the time. This means that a large number of deletions is happening in the instance and that the data is frequently updated. During the examined period, all the experiments read more data than they wrote, but not by a large margin. ATLAS had 2-3 times as much read volume as the written one.

Since quite often the amount of read bytes was significantly smaller than the total size of the file, statistics have been added that show which fraction of the files is read on average. For ATLAS these numbers reach approximately 80-90%. The low numbers could have a negative impact on caching, since the whole file has to be copied to cache but only small fraction of it will be useful. [1]

The metric "Repeated Read Volume" is one of the indicators of how efficient the storage space is used and, at the same time, it can show the potential for providing caching policies in the system. In Figure 2.4 the "Read Volume" is the total volume of all the accessed files and the "Read Workload" is the sum of all the bytes read. The hatched parts show the fraction of the volume that was read more than once and the corresponding fraction of the workload. Some read accesses did not read the files fully, though the average read completion per file at the ATLAS instance is 95.84%.

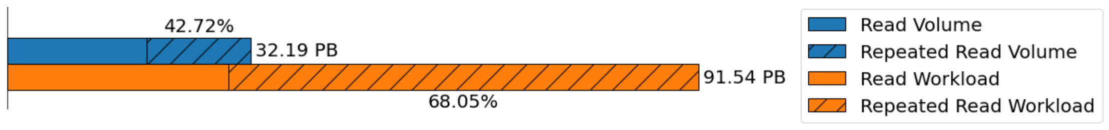


Figure 2.4: Comparison of "Read Volume" and "Read Workload" in the ATLAS Instance, Jan-Mar 2020. The chart presents a breakdown of the data read activities within the ATLAS experiment's computing instance at the CERN T0 Data Center. The "Read Volume," represented by the solid blue bar, stands at 32.19 petabytes (PB) and denotes the total volume of all accessed files. Within this, the hatched blue portion, accounting for 42.72%, indicates the Repeated Read Volume, signifying the volume of the part of the data that was read more than once. The solid orange bar signifies the "Read Workload," the aggregation of all bytes read, amounting to 91.54 PB. The hatched orange section shows the Repeated Read Workload, representing 68.05% of the total Read Workload, which corresponds to the volume of bytes read from files that were accessed multiple times.

2.3.1 Distribution of File Types (Based on Creation/Deletion time)

The access records have been grouped by the file identifiers to obtain file-specific statistics. After, the corresponding files have been categorised based on their creation and deletion

[1]The XCache system, used in the community, tries to address this by a strategy of caching only blocks that have been read and a configuration dependent number of additional consecutive blocks. However, in practice no major differences have been observed

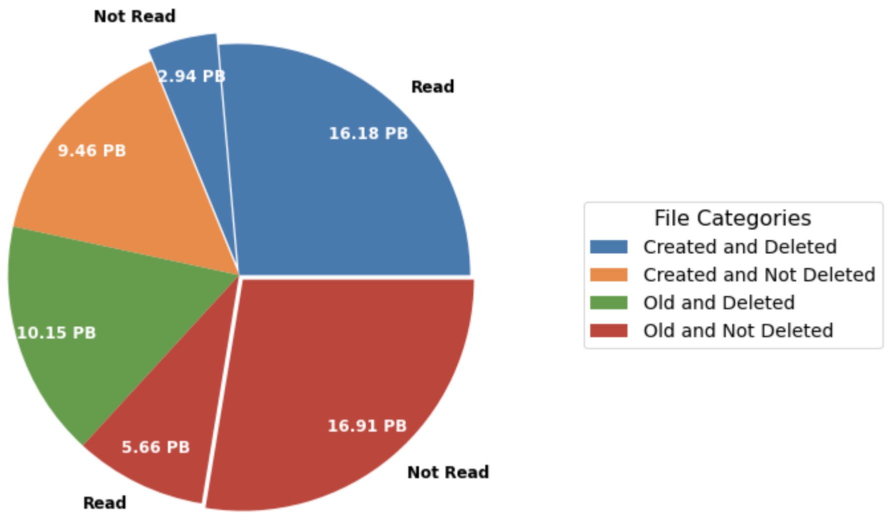


Figure 2.5: The chart presents a breakdown of data read activities within the ATLAS experiment's computing instance at the CERN T0 Data Center, Jan-Mar 2020. Four categories are defined: (1) files created and deleted within the period, which are further classified into "Read" if accessed and "Not Read" groups based on whether they were accessed; (2) files created during the period but not deleted, or deleted subsequently; (3) files created before and deleted within the period, reflecting interim data management practices; and (4) files present both before and after the period, accounting for the largest fraction, also subdivided into "Read" and "Not Read".

times in relation to the boundaries of the considered period. Overall, there are four categories covering all the possible cases:

- files created and deleted during the period;
- files created during the period and deleted after or not deleted;
- files created before and deleted during the period; and
- files created before and not deleted or deleted after the period.

The pie chart in Figure 2.5 shows the distribution of these categories for the ATLAS EOS instance.

The biggest fraction belongs to the files that were present on disk before and after the considered period (36.8% of total volume). For these files, the lifetime is more than three months. The ATLAS experiment produced and deleted approximately the same volume in this period, as a result, the total occupied volume did not change. A big fraction of created files were also deleted (66.9% of created volume), which indicates that there are a lot of short-lived files with a lifetime shorter than 3 months. The fraction of files that went through their whole life cycle ("Created and Deleted") is only 31.2%, and, in the future, the plan is to extend the time frame in order to increase the relative number of such files.

ATLAS jobs produce also a large number of log files and some of them are never read before their deletion. Overall, in the considered period, almost 3 PB of files were not read between their creation and deletion. Moreover, a big fraction of files stayed on disk without being accessed for a long time (refer to Figure 2.5). The possibility of keeping these files on a less expensive storage media should be investigated further [32].

2.3.2 File Sizes Distribution

Another metric of interest for this research is the distribution of file sizes. To visualise this, a plot illustrating the relationship between the number of files and their respective sizes has been created (refer to Figure 2.6a). This plot covers only files that have been accessed at least once during the examined period. There is a clear peak in the distribution around 1 GB, with a maximum of 470.73 GB and an average file size of 875.45 MB.

When comparing Figures 2.6b and 2.6c, most of the volume comes from files bigger than 1 GB, but the majority of files are less than 1 GB. This is yet an indicator that the diversity of file sizes is not negligible, and when looking for appropriate caching models, the file size should be taken into account.

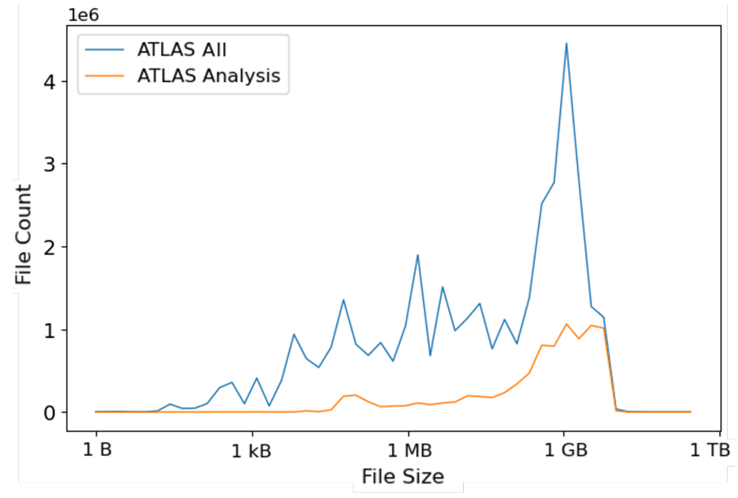
2.3.3 File Accesses Time Distribution (Temporal Locality)

Commencing the exploration, it is crucial to establish a fundamental concept: file popularity. File popularity denotes the frequency with which a file is accessed (read) over a specified period. This metric provides valuable insights into the usage patterns of files within a system, shedding light on the distinction between frequently accessed files and those that remain dormant.

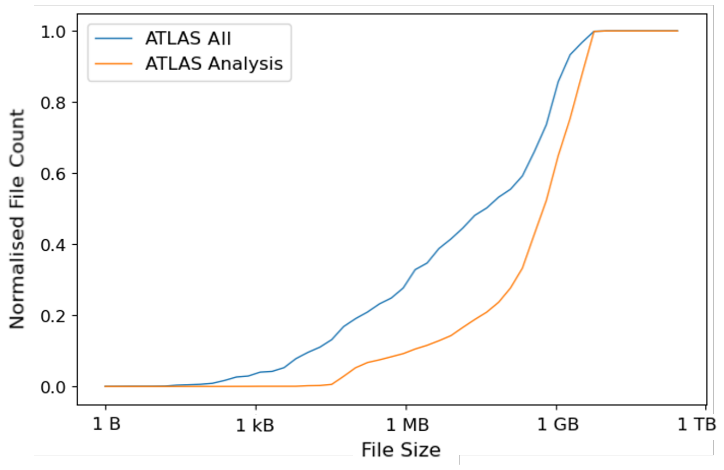
Henceforth, file read access will simply be referred to as file access, considering the minimal occurrence of file modifications within the system.

In this respect, the time differences between consecutive file accesses have been explored; for each read access, the time between this read and the first file access has been calculated.

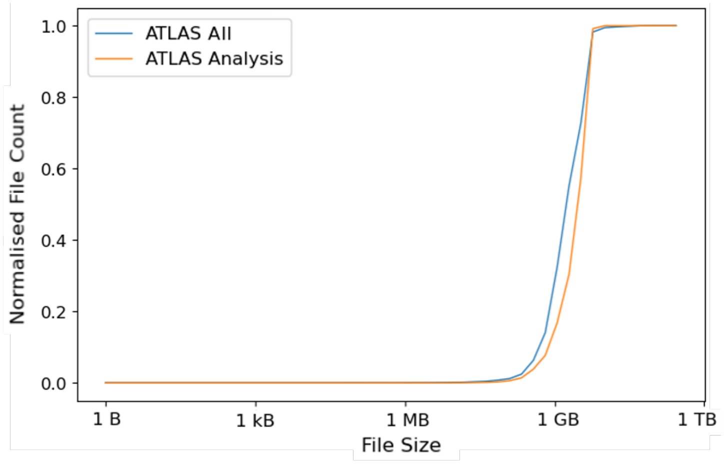
As seen in Figure 2.7, most files were accessed very few times ($\sim$63% of files were accessed only once), and, if a file was re-accessed, it was most likely to happen within a couple of hours (refer to Figure 2.8).



(a) File size distribution.



(b) Cumulative file size distribution.



(c) Cumulative volume contribution by file size.

Figure 2.6: Visual analysis of file size distribution and volume contribution at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. Figure (a) charts the total count of files by their size, distinguishing between all files and those used by the physics analysis activities. Figure (b) demonstrates the cumulative distribution of these files, revealing how file sizes accumulate proportionally. Figure (c) further explores the volume contribution by size, showing the significant data storage footprint that larger files have within the system. Figure (c) was obtained by weighting Figure (b) with file size.

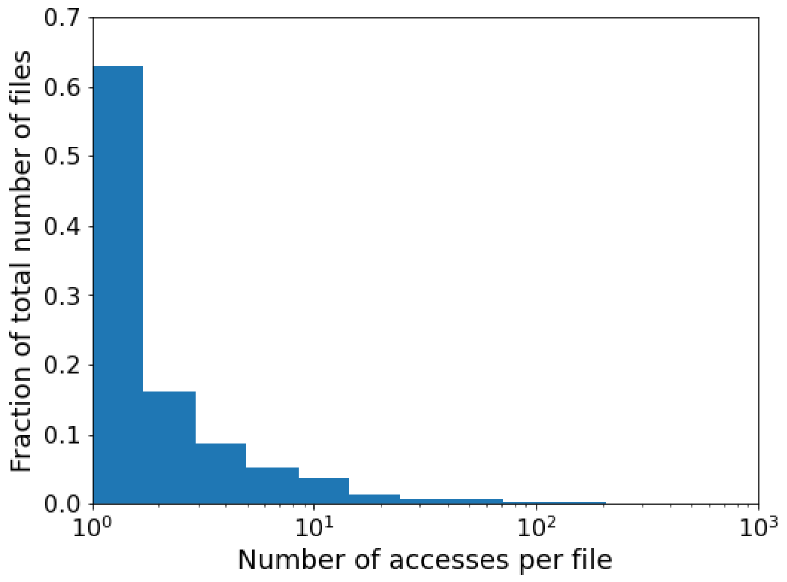


Figure 2.7: Distribution of file access frequency at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram shows the fraction of total files plotted against the number of accesses per file.

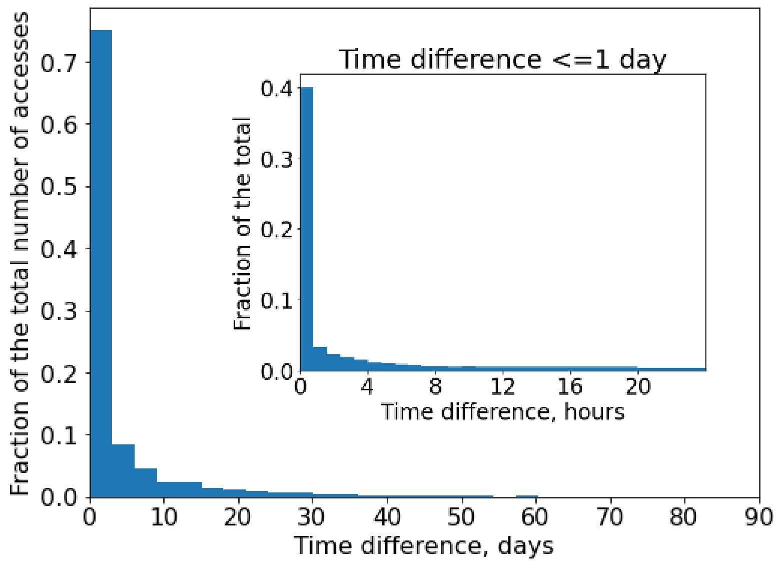


Figure 2.8: Time difference between consecutive accesses at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram displays the fraction of total file accesses by the time interval between consecutive uses, spanning up to 90 days. The main chart illustrates the distribution over days, while the inset focuses on the first 24 hours, highlighting the high frequency of accesses within shorter intervals.

A closer look at the temporal locality of this trace reveals short-term correlations in the request process. Figure 2.9 is derived from plot 2.8, with a log scale applied to the x-axis and y-axis normalisation. The distribution exhibits a multi-modal pattern. While the average time between two consecutive accesses to the same file is approximately 3 days, some files experience consecutive accesses with up to a 60-day gap. Notably, 27.44% of consecutive accesses occur within a minute, and 6.47% occur within one second.

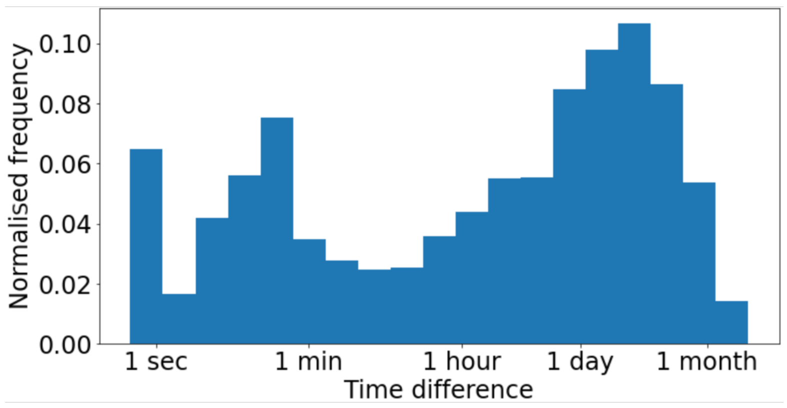
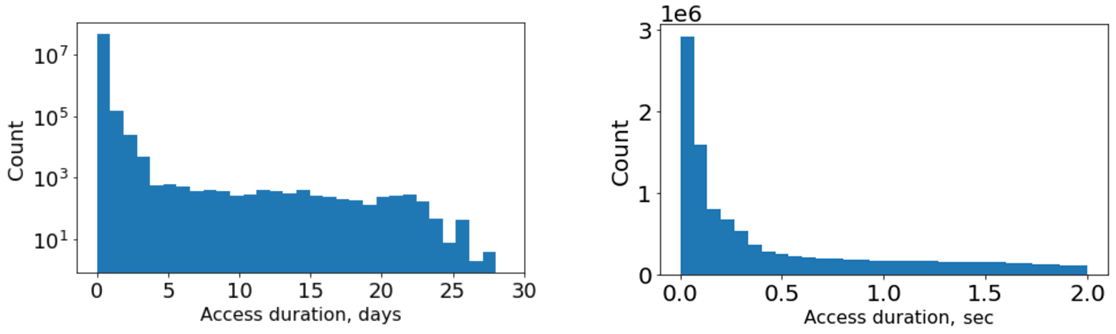


Figure 2.9: Detailed distribution of time intervals between consecutive file requests at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram further analyses the temporal locality of file accesses, highlighting the normalised frequency of time intervals between consecutive requests for the same file, ranging from one second to one month. The distribution is displayed on a logarithmic scale to accentuate the multi-modal nature of file access patterns.

It is essential to acknowledge that some of the observed accesses are not independent read attempts but rather artefacts resulting from the way the experiments access the data, opening the file before working with it to ensure accessibility, analogous to the "touch" functionality in operating systems.

The observations in Figure 2.10a, along with its detailed counterpart focusing on accesses of less than two seconds in duration (refer to Figure 2.10b), corroborate this assertion.

At this juncture, it was decided to proceeded to analyse the trace in its current state, given the focus on optimising the storage system rather than delving into the intricacies of the experiment workflows. This should not significantly impact caching performance; if these artificial accesses occur, the file is typically used shortly thereafter. Nevertheless, it is prudent to be mindful of this when encountering unusual patterns of accesses occurring within one second.



(a) Duration of file accesses.

(b) Duration of accesses < 2sec.

Figure 2.10: Distribution of file access durations at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. Panel (a) shows the frequency of file access durations spanning up to 30 days, illustrating a steep decline in occurrences as duration increases. Panel (b) provides a focused view on shorter access durations less than 2 seconds.

2.3.4 Analysis of Content Popularity and Zipf's Law in Caching Performance

This study was performed for all the 4 detector-experiments at CERN and employs the principle of the Zipf distribution to model content popularity and access frequency within the system. The Zipf distribution is particularly useful in highlighting how a few items dominate usage or popularity in a given system.

To analyse the content popularity, these files have, for a start, been ranked from the most to the least popular. An aspect of critical importance is the tail of the popularity distribution, which can significantly impact caching performance. The analysis of the slope of this tail on a log-log plot (logarithm of requests on the y-axis and logarithm of rank on the x-axis) provides insights similar to the parameters of an equivalent Zipf distribution. This slope essentially corresponds to the Zipf coefficient, which is employed to assess the extent to which Zipf's law characterises content requests in the system.

The results (refer to Figure 2.11), however, reveal a notable deviation from the typical Zipfian distribution observed in web caching scenarios [72]. In contrast to the expectations set by Zipf's law, where a small number of items are vastly more popular, the findings suggest a different pattern of access. Specifically, the Zipf coefficient determined for LHCb is approximately 0.5, whereas for other systems, it hovers around 0.8, more akin to internet request patterns. This divergence emphasises that the system's access patterns are fundamentally different from those typically observed in standard web caching tasks.

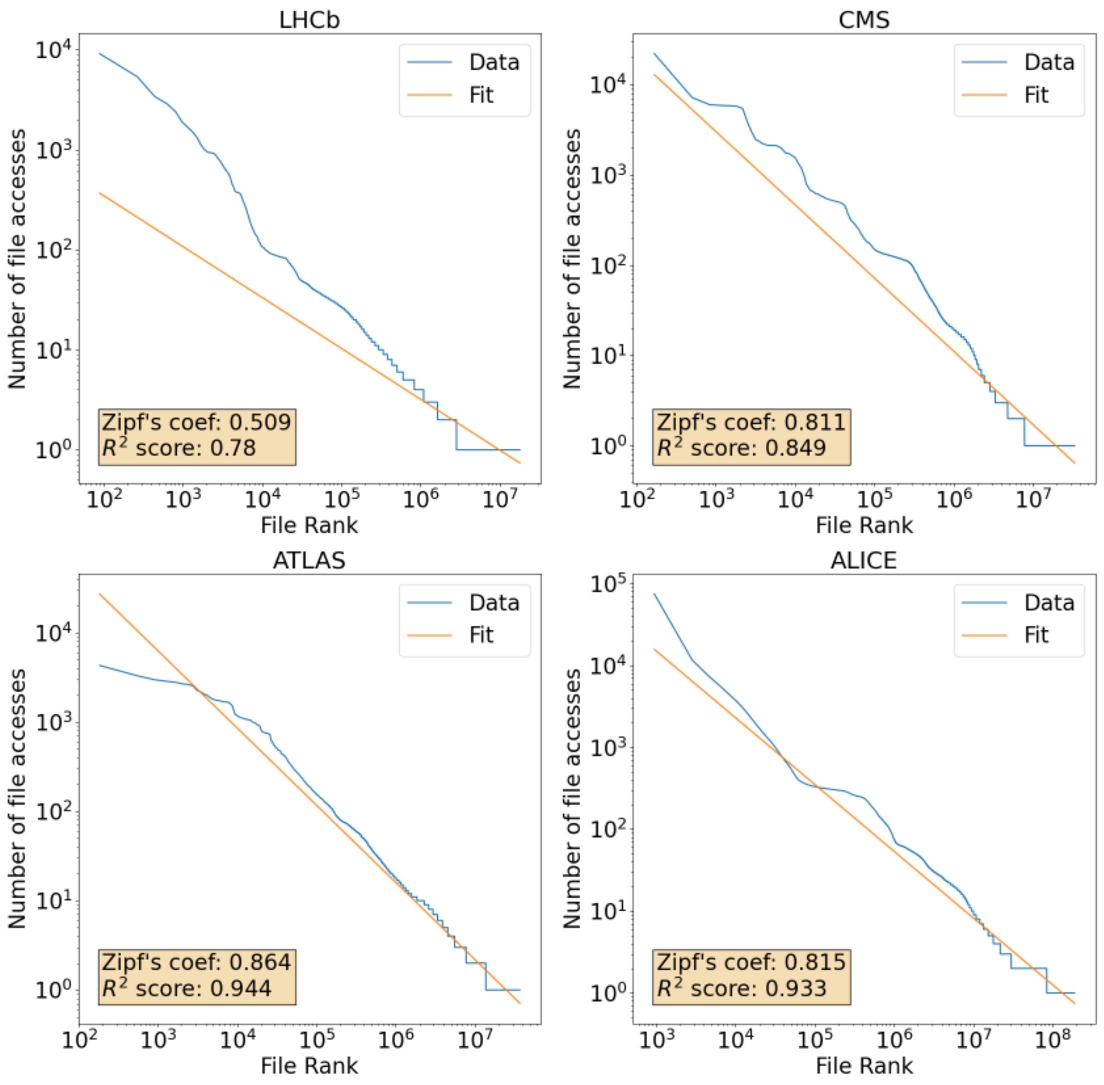


Figure 2.11: Zipf-like distribution analysis of file accesses across different experiments at CERN's T0 Data Center, Jan-Mar 2020. This set of plots displays the ranking analysis of file accesses for the LHCb, CMS, ATLAS, and ALICE experiments, comparing empirical data (blue) against a fitted Zipf's Law model (orange). Each graph shows the number of file accesses plotted against the file rank on a logarithmic scale. The Zipf's coefficient and R^2 scores are annotated, indicating the degree of alignment between the data and the Zipfian distribution.

2.4 Analysis Files Access Patterns

A typical WLCG data processing workflow could be roughly divided into preprocessing (reconstruction, derivation, etc.) and analysis activities. The preprocessing campaigns are run in a well-organised, scheduled manner. Therefore, provisioning of resources for the preprocessing activities could be done in advance since they are more or less predictable. On the other hand, the analysis is aimed at preparing the RAW data coming from the LHC detectors for the physics users. These analysis tasks are relatively sporadic as they are independently triggered by different users according to their needs. Consequently, they are not as scheduled and organised and less predictable.

Analysis jobs constitute only a fraction of the total EOSATLAS node workload, but these activities are not as scheduled and organised as other types of jobs (reconstruction, derivation, etc.) and, consequently, less predictable. These kinds of jobs work with special data formats (AOD and DAOD files). For these cache studies, therefore, they are addressed separately.

Isolating caching exclusively for analysis data is a viable option to consider. This research would be particularly useful for the processing T2 sites that have temporary storage only for the analysis data.

2.4.1 Workload Comparisons: Analysis vs. Total Workloads

Analysis jobs constitute only a fraction of the total EOS ATLAS workload. Table 2.6 provides an overview of the read workloads for analysis files compared to the total workloads. It is evident that analysis files contribute to approximately one-quarter of the total number of file accesses/files and approximately 60% to the total access workload/volume. Interestingly, these percentages remain nearly unchanged in the actual sequence of requests. This also leads to a preliminary assumption that, on average, analysis files are larger than the rest. To substantiate this, a more detailed study of the distribution of file sizes has been conducted (refer to Section 2.4.2).

2.4.2 Comparing File Sizes: Analysis Files vs. All Files

Figure 2.6a illustrates the size distribution of files that were accessed during this period (not necessarily encompassing all the files present on the instance) and provides a separate view of the size distribution for the analysis files. In general, the size of the accessed files spans from a few bytes to half a terabyte, with the most prevalent file size being around 1 GB, and an approximate average size of 875 MB. Conversely, for the analysis files, there are fewer smaller files, with the peak size at around 10 GB. Although, when

Metric	**Total**	**Analysis**
Number of Read Accesses	173,181,554	45,931,029 (26.5% of total)
Accesses Volume	91.54 PB	55.46 PB (60.6% of total)
Number of Accessed Files	36,774,178	9,152,849 (24.9% of total)
Volume of Accessed Files	32.19 PB	19.1 PB (59.3% of total)

Table 2.6: Comparison of the total access and analysis-only access workloads within the ATLAS experiment's computing instance at the CERN T0 Data Center, Jan-Mar 2020. "Number of Read Accesses" refers to the count of read access events recorded. "Accesses Volume" denotes the cumulative data volume accessed, measured in petabytes (PB). "Number of Accessed Files" indicates the total count of individual files accessed. "Volume of Accessed Files" represents the aggregate file size of accessed files, also measured in petabytes. The percentages in parentheses show the fraction of each metric that corresponds to the analysis-related files or workloads compared to the total.

compared to the entirety of files, the maximum file size is smaller (approximately 70 GB), the average remains higher (2 GB), reaffirming the earlier assumption.

Figure 2.12 presents the sizes of AOD and DAOD files separately, indicating that the dispersion in the distribution is influenced by both file formats.

2.4.3 Number of File Accesses

The popularity of the analysis files demonstrates significant diversity (refer to Figure 2.13). Upon closer examination, it becomes evident that while a large proportion of files are accessed infrequently, there remains a substantial number of files that are re-accessed multiple times. This characteristic underscores the potential benefits of caching within the system and reinforces the decision to explore caching strategies for optimisation.

2.4.4 Correlation Between File Size and Popularity

Furthermore, the distribution of file popularity, as indicated by the number of accesses, is not uniform across different file sizes, as evident in Figure 2.14. Although there is no direct correlation between these two metrics, it's noteworthy that very small files (less than 10 kB) and very large files (greater than 1 GB) tend to experience higher access rates compared to files of average size. Figure 2.15 provides an insight into the cumulative number of file accesses relative to file size. It becomes apparent that the majority of accesses are concentrated around files of approximately 1 GB in size. Consequently, the system load, depicted in Figure 2.16 and calculated by weighting the previous plot with file sizes, is predominantly influenced by files falling within the range of 100 MB to 10 GB.

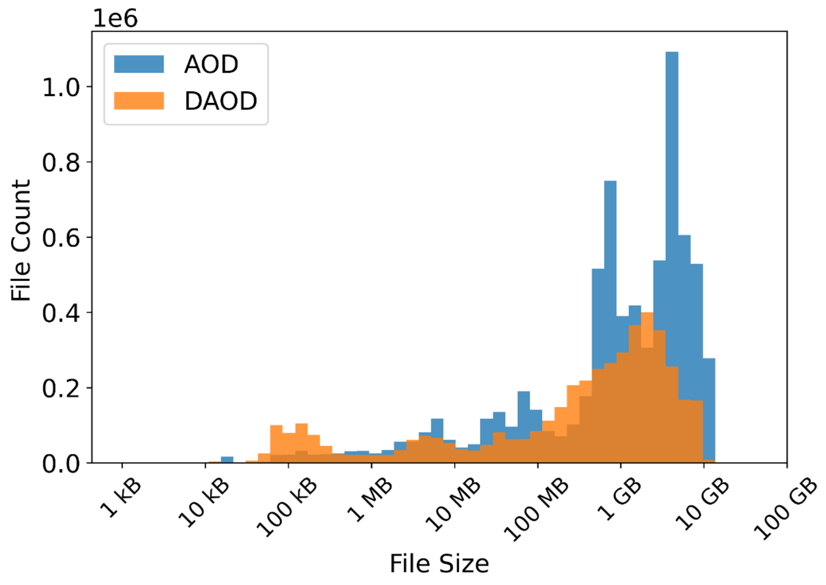


Figure 2.12: File size distribution for AOD and DAOD Files at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram illustrates the distribution spread and the differences in file sizes between AOD and DAOD, categorised from 1 KB to 100 GB. The distribution shows the count of files for each size category, highlighting the predominant file sizes for each format.

Based on the observations above, it is evident that file size cannot be overlooked in this analysis. While very small files contribute significantly to the overall number of accesses, it is the larger files that predominantly contribute to the system load.

2.4.5 Request Rate

Another significant characteristic of the trace is the variability in request load over time, as depicted in Figure 2.17. The byte rate, averaged over one second, exhibits a wide range, ranging from a few bytes to several terabytes per second, with the mode of the distribution hovering around 20 GB/s. Additionally, the average request rate throughout the entire period is recorded at 9.91 GB/s.

2.4.6 Dependency Between the Lifetime and Popularity

Next, the investigation focused on exploring any potential correlation between the lifetime of a file and the frequency of its accesses. The findings are illustrated in Figure 2.18a, revealing that files with longer lifespans tend to accumulate a higher number of accesses. This observation led to the hypothesis that the frequency of access per unit of time might

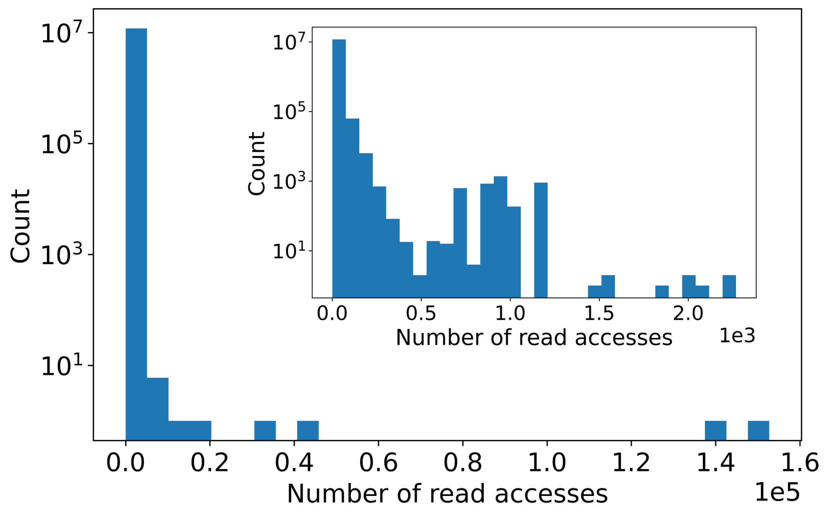


Figure 2.13: File popularity distribution for analysis files at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram shows the distribution of the number of read accesses per analysis file, illustrating file popularity across a broad range of counts from a few to over a thousand reads. The main chart provides a broad overview, with the inset offering a detailed view of the distribution for files with up to 2,000 reads.

exhibit uniformity throughout the files' lifespans, a concept further substantiated by the data presented in Figure 2.18b.

2.4.7 Popularity of the Files: Ranking-Zipf Plots

In the next step of the investigation follows the examination the Zipf-based qualities of the analysis files in comparison to other files within the system. While these two categories exhibit slightly different distributions, neither adheres strictly to the conventional patterns suggested by Zipf's law. Despite this deviation, both groups of files possess relatively high Zipf scores: 0.95 for analysis data and 0.94 for the rest. This finding suggests a unique characteristic in the way content is accessed in the system, differentiating it from typical models anticipated by Zipfian distributions.

2.5 Dataset-Based Access Patterns

Consider that all the records meaningful for the same physics analysis are spread across different files and together form a logical entity called a dataset. The properties of datasets and their access patterns have been studied in the context of this thesis. The distribution of dataset sizes is more heterogeneous than that of individual files (Figure 2.20). The

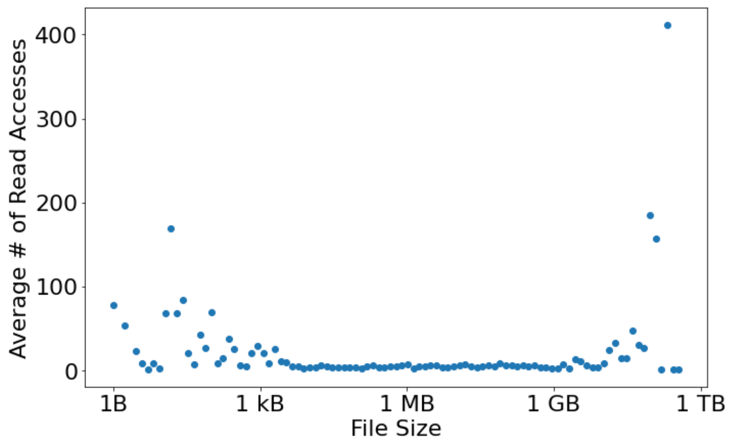


Figure 2.14: Correlation between file size and average number of read accesses at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This scatter plot illustrates the average number of read accesses per file as a function of file size, ranging from 1 Byte to 1 Terabyte.

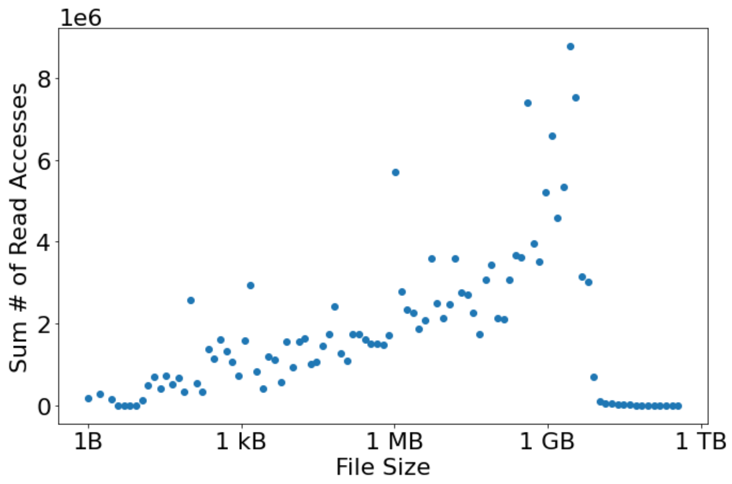


Figure 2.15: Total number of read accesses by file size at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This scatter plot shows the cumulative number of read accesses for files across various sizes, from 1 Byte to 1 Terabyte.

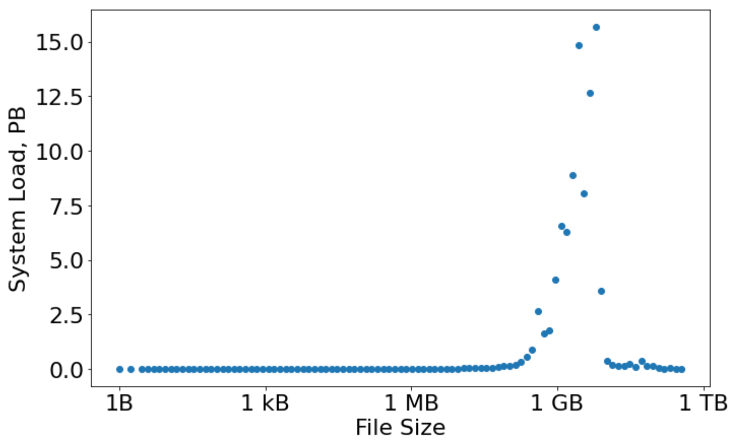


Figure 2.16: System load correlated with file size at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This scatter plot depicts the system load, calculated by weighting the total access count with file size and measured in petabytes, as a function of file size, ranging from 1 Byte to 1 Terabyte.

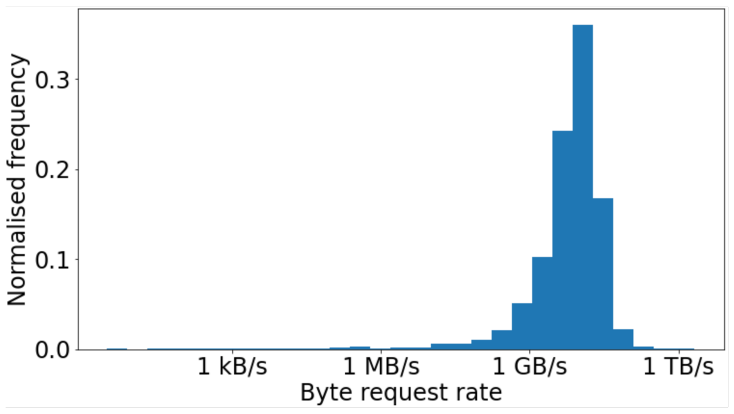
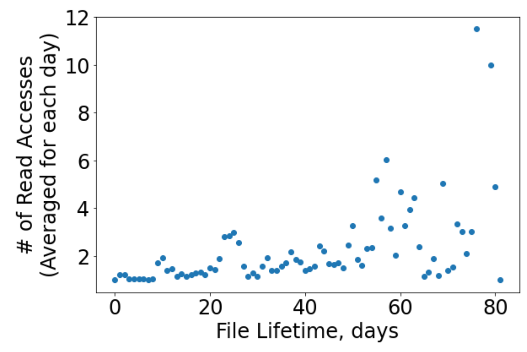
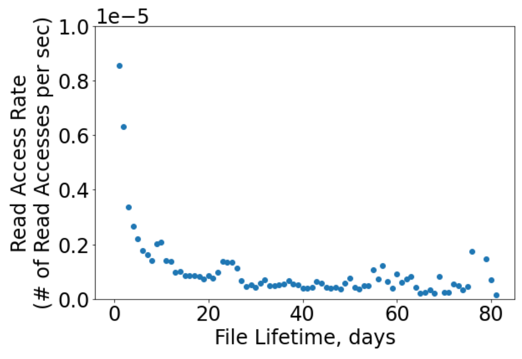


Figure 2.17: Distribution of byte request rate averaged per second at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram displays the normalised frequency of byte request rates, ranging from 1 KB/s to 1 TB/s.



(a) Number of file accesses depending on the lifetime.



(b) The access rate depending on the lifetime.

Figure 2.18: Dependency between file lifetime and number of accesses at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This figure consists of two panels: (a) shows the average number of file accesses per day against the file's lifetime in days, and (b) depicts the read access rate per second as a function of the file's lifetime. Both plots illustrate how the frequency of file usage varies over the lifespan of files.

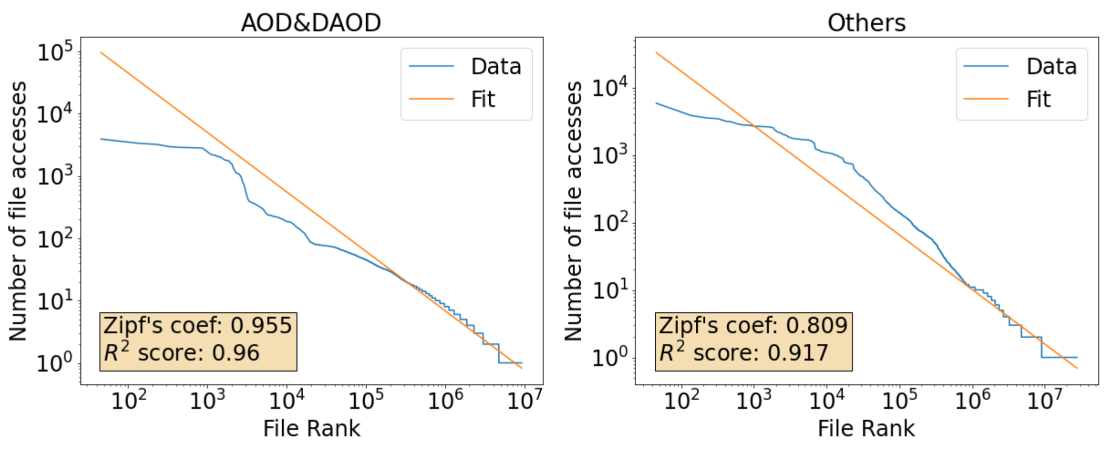


Figure 2.19: Comparative analysis of Zipf-like distributions for analysis and non-analysis file accesses across the ATLAS experiment at CERN's T0 Data Center, Jan-Mar 2020. This figure compares empirical data (blue) against a fitted Zipf's Law model (orange) for two categories of file accesses: analysis ("AODDAOD") and non-analysis ("Others"). Each graph shows the number of file accesses plotted against the file rank on a logarithmic scale. The Zipf's coefficient and R^2 scores are annotated, indicating the degree of alignment between the data and the Zipfian distribution.

most common dataset size is around 1 MB, while the average is almost 80 GB.

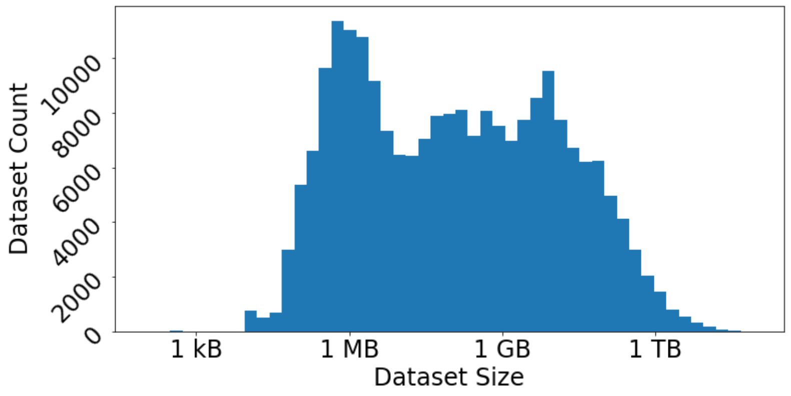


Figure 2.20: Dataset Size Distribution at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram displays the distribution of dataset sizes within the system, ranging from 1 KB to 1 TB.

An intriguing observation from the distribution of files per dataset reveals that the majority of datasets are characterised by their compactness, with most encompassing fewer than 5 files. (Figure 2.21) Here, some outliers and the long tail (5%) are excluded.

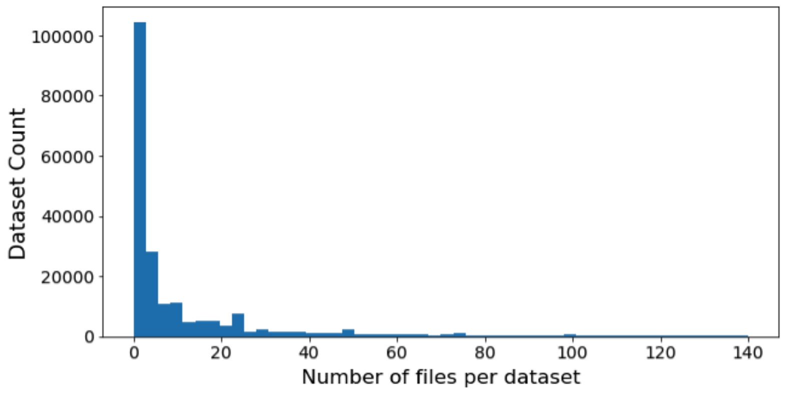


Figure 2.21: Distribution of the number of files across datasets at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram shows the count of datasets against the number of files they contain, up to the 95th percentile.

A notable trend in this data analysis highlights that the majority of datasets experience a limited access frequency, with most being accessed fewer than 20 times during the observation time. (Figure 2.22)

In this part of the research,the focus shifte to determine whether the datasets were utilised fully. To answer this question, it has been examined how many files within each dataset were accessed throughout the three months. Understanding the full utilisation

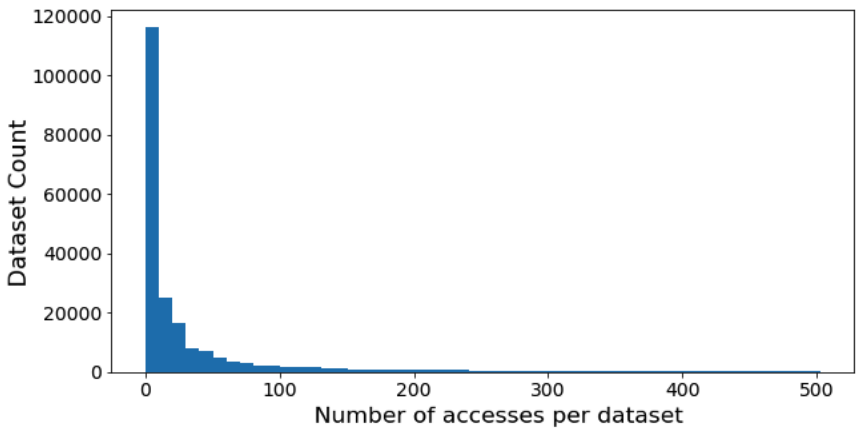


Figure 2.22: Distribution of file access frequencies across datasets at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram represents the number of times files within datasets are accessed, up to the 95th percentile.

of datasets is pivotal for assessing the efficiency and coverage of the data management practices.

The task of determining if the datasets are accessed in their entirety is complicated by the challenge of distinguishing between consecutive runs within the same dataset. The primary focus, however, was to ascertain whether all files within the datasets were accessed at least once during the observed period.

The findings revealed that in 75% of the datasets, every file was accessed at least once. On average, 83% of the files in each dataset were accessed, although it is critical to note that this figure does not imply usage in every individual access instance. Detailed results of this analysis are presented in Figure 2.23.

To determine whether this pattern occurs consistently with every dataset access, both the mean and variance of the number of file accesses per dataset have been computed. The results are depicted in Figure 2.24. Notably, the variance tends to be quite low, suggesting that the frequency of file access remains relatively consistent within datasets.

The order of the individual file accesses is difficult to reconstruct since log files only show the aggregate request process, where multiple users may access the same dataset simultaneously, and the same user can access different parts of the same dataset in parallel (for example, when the data being processed is coordinated by Rucio).

From all the above, it is safe to assume that when a dataset is accessed, almost always all of its files are indeed accessed. However, the order of access within the dataset may not follow a specific sequence.

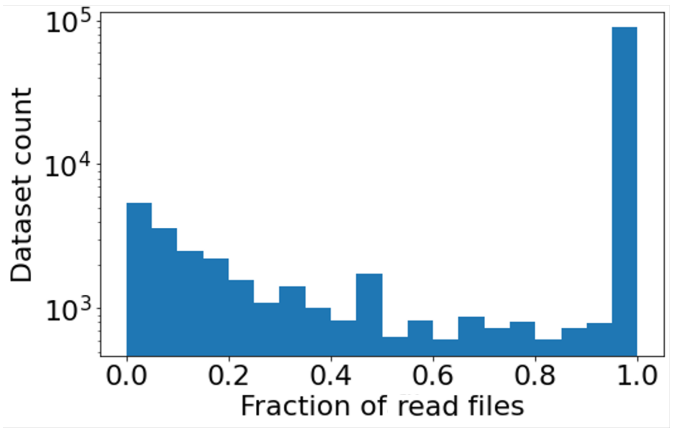


Figure 2.23: Histogram of file access proportions across datasets at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This graph displays the dataset count on a logarithmic scale against the fraction of files within each dataset that have been accessed.

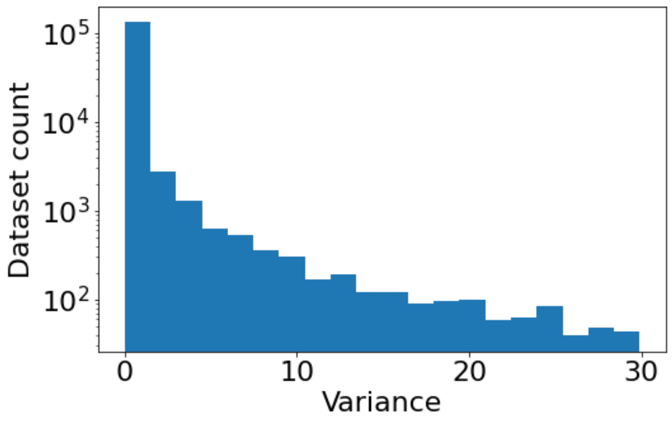


Figure 2.24: Variance in file access frequencies within datasets at CERN's T0 Data Center, ATLAS experiment, Jan-Mar 2020. This histogram shows the distribution of variance in the number of file accesses per dataset, plotted on a logarithmic scale for dataset counts. A high frequency of low variance values indicates uniformity in access across files within most datasets.

2.6 Conclusions

In this chapter, an analysis of a comprehensive data access trace characterised by a broad spectrum of file sizes and diverse file popularity trends has been presented. This trace serves as an excellent basis for assessing cache performance and exploring the efficacy of various cache eviction strategies.

Throughout the data handling and processing layers, Rucio and PanDA consistently log operations on files, tasks, and datasets. When this data is merged with the access logs from the EOS file system, a holistic view of the data's entire lifecycle emerges, from its global distribution to the specific file usage patterns. This analysis offers vital insights, laying the groundwork for further research to enhance storage management and utilisation. Although expanding this research to encompass longer durations, additional storage infrastructures, and more comprehensive details would be beneficial, the current findings still provide a solid foundation for future optimisation efforts. Moreover, the methodologies developed are versatile enough to be applied in broader contexts beyond the ATLAS project.

The investigation into AOD and DAOD datasets reveals that most data is accessed infrequently, with access typically occurring in short bursts, and a substantial portion remains unused on disks for prolonged periods. These preliminary findings, while not universally applicable to all experiment stages, suggest that implementing more robust caching, adopting stricter data deletion policies, and varying the quality of service levels could significantly reduce storage costs. However, realising these benefits in practice necessitates further work, particularly in ongoing data popularity monitoring and the creation and application of detailed metrics to guide both users and site managers toward more resource-efficient practices.

The access patterns of analysis files differ remarkably from the general trends in file size and popularity. This distinction underlines the need for differentiated optimisation strategies for grid site management as opposed to optimising experimental analysis workflows. The focus is on the latter, addressing the challenge of storing and processing analysis files, which are distributed across WLCG sites. It is crucial to recognise that local file storage and access patterns at individual sites can vary considerably based on factors like the site's role and geographic location.

Another key observation from this analysis are the varied file size distribution within the dataset. These findings underscore the importance of considering this diversity in designing caching models, emphasising the need for strategies that can adapt to a wide range of file sizes.

2.7 Log Data Processing Pipeline: A Brief Overview of Implementation Details

Overall, the developed data processing pipeline encompasses a series of steps, starting with collecting data from two primary sources: EOS Report Logs and Rucio Logs dumps. The size of EOS Report Logs ranges between 100 MB to 8 GB daily for each experiment.

In parallel, data from Rucio Logs dumps has been integrated, sourced from the Rucio database. These logs aggregate approximately 50 GB over three months.

After the main log processing stages (parsing, filtering, grouping, merging) the logs are converted into .parquet files. This format not only reduces the data size by about 30 times (resulting in files ranging from 600 MB to 7 GB monthly) but also makes the data more manageable for statistical analysis and visualisation.

Furthermore, "ordered traces" have been constructed from the processed logs, tailored to suit individual Cache Eviction Policies. These traces, stored as .csv files and organised by timestamp, occupy 2 to 6 GB over a three-month span.

Both processed logs and ordered traces are stored in HDFS and EOS, ensuring streamlined access and retrieval.

The final component of the pipeline is the implementation of diverse cache eviction policies. Utilising languages like C++, Python, and Python with PySpark allows for an exhaustive evaluation of caching strategies.

Chapter 3

Caching in the Context of the WLCG

3.1 Problem Statement

To formalize the problem of searching for an optimal cache, the total number of requests (or the trace length) is defined as N, M is the number of unique files in the trace, C is the cache capacity, $f_1, f_2, \ldots, f_M$ are the unique files with sizes $s_1, s_2, \ldots, s_M$ respectively, V is the total unique volume requested ($V = \sum_{i=1}^{M} s_i$). The sequence of requests can be represented as $f_{j_1}, f_{j_2}, \ldots, f_{j_N}$, where j_i is the file requested at time $t_i = i$ (t_i therefore changes from 1 to N).

The result of applying a given cache eviction policy to the trace can be represented as a sequence $h_1, h_2, \ldots, h_N$ with $h_i \in \{0, 1\}$, where $h_i = 1$ if the corresponding file f_{j_i} was retrieved from the cache (a hit) and $h_i = 0$ otherwise (a miss). In this case, $N_{cache} = \sum_{i=1}^{N} h_i$ is the number of file retrievals from the cache (the total number of hits), and $V_{cache} = \sum_{i=1}^{N} h_i \times s_j$ is the volume (the total amount of bytes) retrieved from the cache.

3.2 Constructing *MRCs* for Equal-size File Traces

A *Miss Ratio Curve* (*MRC*) is a graphical representation that serves to elucidate the relationship between cache performance and cache size. It provides valuable insights into how the cache hit ratio changes as the cache size varies. In this study, *MRCs* were created for the historical trace to understand how cache performance varies with different cache sizes. Two types of *MRCs* will be important to distinguish in this research: those representing *FMR* and *BMR*.

To start, studies have been conducted with request traces consisting of a sequence of file IDs accessed. At this stage the actual times between accesses or the file size haven't been considered, this is a significant simplification, given the range of file sizes in the instances. After obtaining initial results and gaining an understanding of the computational complexity and the general shape of *MRCs*, work shifted to more complete versions of the traces containing file sizes as well (refer to Section 3.4).

3.2.1 *OSA*. Samples

To construct *MRCs* for *Optimal Static Allocation* (*OSA*), one needs information about file popularity (here the notion of popularity corresponds to the number of times a files was accessed). Therefore, historical data is needed from which file popularity can be derived.

For a cache of size C, the policy would dictate to put C of the most popular files into the cache. The *File Hit Ratio* can then be simply calculated as a ratio of the total number of requests to the C most popular files to the total number of requests. In practice, when working with a tabular data format, this could be done by sorting files by popularity columns and calculating the cumulative sum of these popularities, starting from the most popular file.

The first approach was to work with a trace sample of size 100,000 files (a sub-sequence of the full trace with only the requests to a previously randomly selected set of 100,000 files). In this case, it was feasible to do the calculations described above using Python `pandas` dataframes which allowed rapid experimentation to gain insight [73].

Following a similar logic, it was possible to calculate *Byte Miss Ratio* curves, where the *Byte Hit Ratio* is determined as a ratio of requested bytes that were served from the cache to the total volume requested.

3.2.2 *LRU* and *OPT*. Samples

LRU is one of the simplest cache eviction policies to construct *MRCs*. It satisfies the inclusion policy that states that in each moment in time, a cache of size C is a subset of a cache of size $(C + 1)$ for all C. This allows for the construction of a so-called cache stack: at each point t in time, a subset $\mathit{cache\_stack}[1 : C]$ corresponds to the files that would be stored in a cache of size C after processing t first requests. Here, the notion of time corresponds to the position in the request trace. This allows to find stack distances for each file request, the minimum cache size is given by the size for which the requested element would be found in the cache if it was ever accessed before. An array of stack distances can be easily converted into the *MRC*.

This logic was inspired by the ideas presented in [37].

This publication suggests an algorithm to calculate stack distances for *OPT*, the *Optimal Policy* for equal-sized file trace. This policy, also known as *Clairvoyant* or *Belady*, evicts the file that will be accessed the furthest in the future. The publication also presents proof that this algorithm satisfies the inclusion policy.

Since after each file request the cache stack needs to be re-ordered, the time complexity is $O(N \times M)$ for both *LRU* and *OPT*, even though in practice only a small portion of the cache stack needs to be re-ordered and the average time complexity of each file request is tighter than $O(M)$. Given this complexity, it was possible to process samples only of the order of $10,000$ files.

Even though the time complexity for *LRU* could be improved, by maintaining a tree of files ordered by the next request time using so-called "counting trees", to achieve $O(\log M)$ complexity per request as explained in [74], there is also a proof demonstrating can be found that *LRU* is the only algorithm for which it is possible since the stack ordering [1] for this algorithm is identical to the priority ordering [2].

3.2.3 *OSA*, *LRU*, and *OPT*. Samples. Comparison

To compare the performance of the three algorithms (*LRU*, *OSA*, and *OPT*) on an identical dataset, *MRCs* for *OSA* have been developed, using the same sample of 10,000 files. The corresponding plots are presented in Figure 3.1.

If *LRU* performs better than *OSA*, which is the case for ATLAS and ALICE, there must be a lot of temporal locality. In this case, there is little benefit from static caches.

If *OSA* performs better than *LRU*, as seen for LHCb, there is a small set of very popular files, and even having a static cache with these files yields a lower miss ratio than a simple online cache eviction policy.

Furthermore, regardless of whether *OSA* or *LRU* performs better, a gap exists between the best-performing policy and *OPT*, which represents the theoretical lower bound of what could be achieved. This gap serves as a motivation to develop a new policy that builds upon the strengths of *OSA* or *LRU* to more closely approach the efficiency of *OPT*.

Additionally, these plots show that most of the gain from the cache could be attained with a cache of size 20-40% of the total number of requested files.

[1] the order of files in the cache stack

[2] the order of files based on the next request time

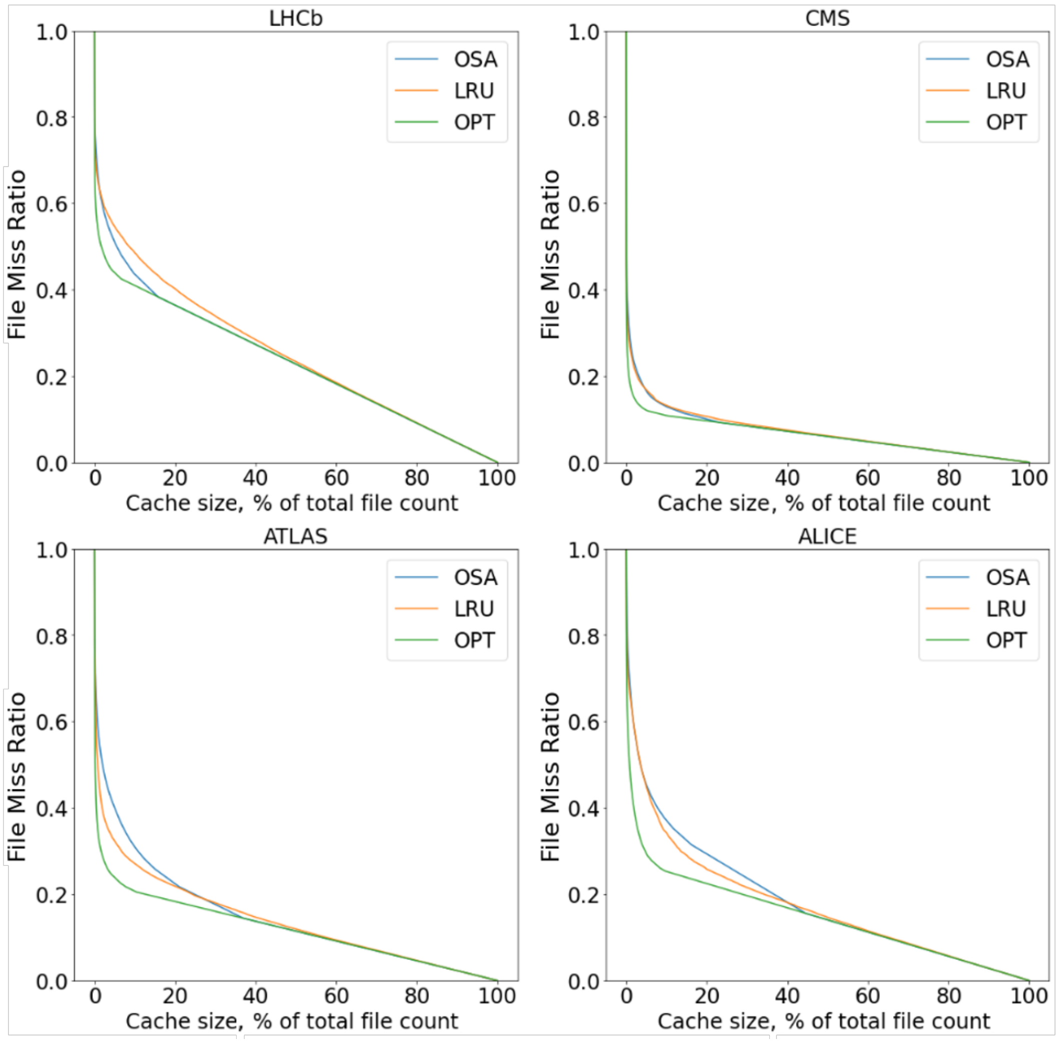


Figure 3.1: Comparative analysis of *File Miss Ratio* across different cache sizes and eviction policies for the LHC detectors. The figures display the *File Miss Ratios* as a function of cache size, represented as a percentage of the total file count, for three different cache eviction policies: *Optimal* (*OPT*), *Least Recently Used* (*LRU*), and *Optimal Static Allocation* (*OSA*). Each line traces the performance of a policy, showing how the Miss Ratio decreases as the cache size increases, from 0% to 100% of the total file count. Cold misses, which occur when data requested has never been in the cache, are excluded from this analysis.

3.2.4 *OSA*. Influence of Sampling

To estimate the influence of sampling on *MRCs*, the case of *OSA* has been studied and the computations in Python have been converted to pySpark to be able to construct *MRCs* on the full traces. Unfortunately, the need to calculate a cumulative sum requires all rows to be brought to a single Spark node, which negates the benefits of the distributed nature of computations in Spark. Moreover, this approach proved infeasible for ALICE, as the number of files (rows) was too large to fit on a single node.

Nevertheless, the *MRCs* that could be obtained in a reasonable amount of time have been compared (Figure 3.2). It can be seen from the plots that samples seem to give a very good approximation, as in the case of LHCb and ATLAS and the 100,000 files sample for CMS, as well as over- or underestimate the miss ratios, for the 10,000 files samples for CMS and ALICE.

The construction of *MRCs* using sampling was also addressed in the previous scientific works. The paper [75] summarises state-of-the-art techniques to eliminate the effect of sampling. They propose a hybrid way of constructing *MRCs* where the part of the curve that corresponds to the small cache sizes is built precisely, as this part suffers the most from sampling, and the rest, corresponding to large cache sizes, is built by compensating sampling algorithms.

Instead, these studies proceeded with constructing *MRCs* by points: calculating precise miss ratios, but only for a finite set of cache sizes. This approach also enabled the analysis of ALICE's data, which included nearly 200 million accesses over three months, achieving faster results with only minimal compromise in accuracy.

3.3 Cache Admission: Impact on Algorithmic Performance

In the preceding discussion, the focus has been primarily on cache eviction policies. However, it is imperative to also consider admission policies, which determine the initial inclusion of a file in the cache. The decision-making process surrounding whether to admit a file to the cache is a critical aspect of cache management.

Each cache policy presents a binary choice regarding the mandatory status of storing an element in the cache. These options significantly influence the performance of the algorithms, leading to varied outcomes based on the chosen policy.

In this section, the differences between two caching policies are examined: one that does not require the storage of the requested file in the cache, denoted as $P(k)$, and another that enforces the constraint of storing the last object requested in the cache, denoted as $P_c(k)$. For simplicity, it is assumed that objects of equal size are considered.

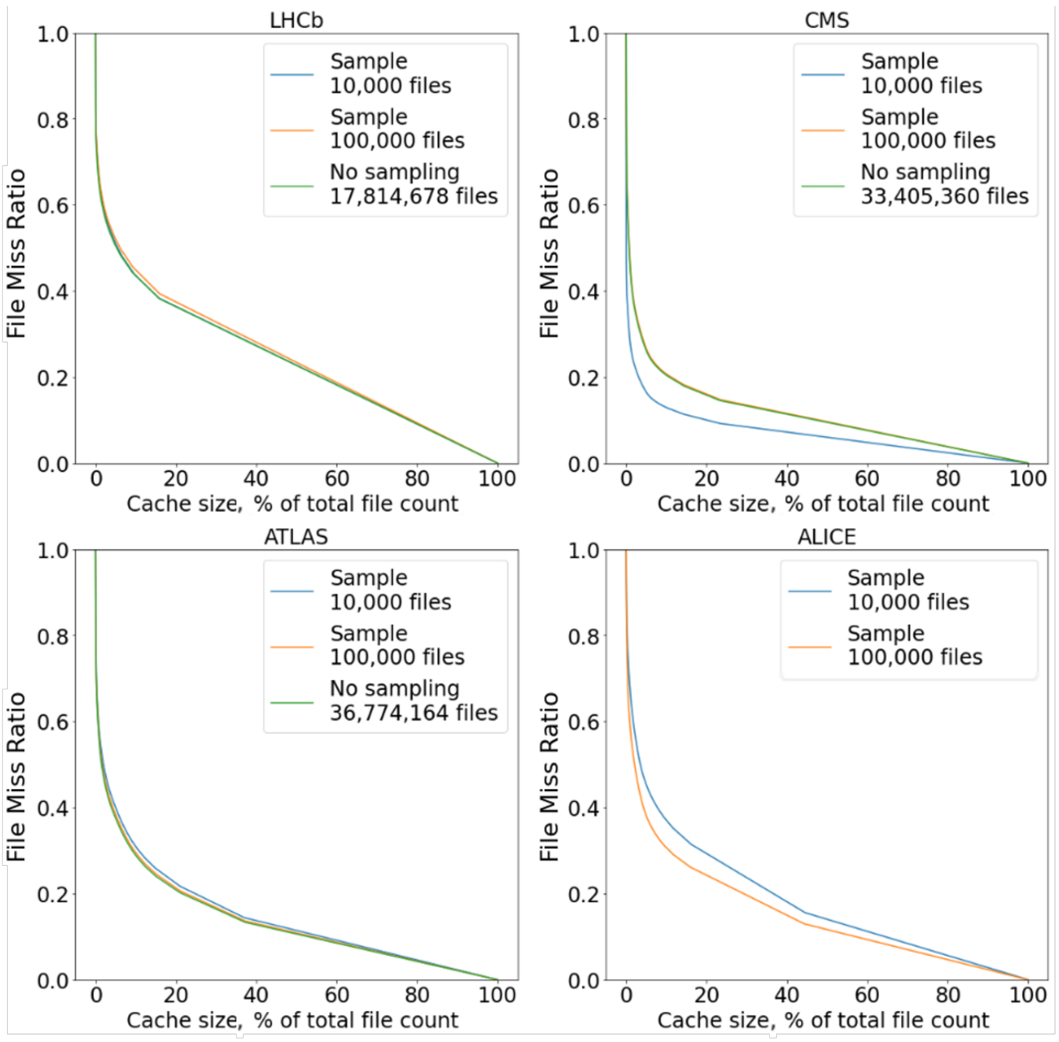


Figure 3.2: Impact of trace sample size on *File Miss Ratio* for the *Optimal Static Allocation* (*OSA*) eviction policy in the LHC detectors. The figure presents the *File Miss Ratios* as a function of cache size, shown as a percentage of the total file count, for the *OSA* eviction policy using different sample sizes. The curves illustrate how the file miss ratio decreases with increasing cache size, evaluated for sample sizes of 10,000 files, 100,000 files, and no sampling. Cold misses, which occur when data requested has never been in the cache, are excluded from this analysis.

Let k represent the cache size. $P(k)$ and $P_c(k)$ are defined as the optimal caching policies without and with the constraint of storing the last object requested, respectively. The metric of interest is the hit ratio.

It is clear that $P(k)$ outperforms $P_c(k)$ in terms of the hit ratio, which can be expressed as $P(k) \geq P_c(k)$. However, when considering the caching policy $P_c(k+1)$, which allows for one additional slot in the cache, it is observed that it performs at least as well as $P(k)$. This can be stated as $P_c(k+1) \geq P(k)$, as $P_c(k+1)$ can store what $P(k)$ stores and reserve the extra slot for the last object requested, if it is not in $P(k)$, or for some other object.

In summary, these two caching policies form a "sandwich" relationship:

$$P_c(k+1) \geq P(k) \geq P_c(k).$$

However, it is important to note that the performance of a cache with size k or $k+1$ is practically indistinguishable for cache sizes not too small, such as $k \geq 100$. Therefore, for cache sizes $k \geq 100$, there is virtually no difference in the results obtained by policies P and P_c:

$$P_c(k) \approx P(k).$$

3.4 *MRCs* for Heterogeneous File Sizes

3.4.1 Optimisation of *MRCs* Through Point-Based Construction

In the following research, the focus has shifted towards a more efficient method for constructing *MRCs*. Initially, the process of fully simulating *MRCs* for each cache size proved to be time-intensive. For instance, using Python's `OrderedDict` for simulating *LRU*, it took several hours for datasets such as LHCb and CMS, and over eight hours for ATLAS.

Furthermore, moving the construction of *MRCs* to select discrete points rather than evaluating every possible cache size was seen as advantageous. This change stems from the understanding that these functions are non-increasing and recognising the need to economize on time spent on this aspect of the study. By transitioning to a point-based visual representation of *MRCs*, sufficient accuracy for comparing different caching algorithms has been achieved, which is one of the central objectives of this research.

This streamlined approach allows for the inclusion of more complex algorithms in the study, the processing of complete access traces, and the consideration of actual file sizes. Now, instead of constructing a comprehensive spectrum of cache sizes with corresponding exact hit ratios, it suffices to simulate the cache for only 10-12 strategic points. For these points, the caching policy is implemented, and a full simulation is run. The performance is then evaluated in terms of *File Miss Ratio* (*FMR*) and *Byte Miss Ratio* (*BMR*).

It is important to note that certain assumptions are still made about the system. For instance, in this section, aspects such as throughput and delayed hits are not considered. These are examined in more detail in Section 3.8.

3.4.2 Lower and Upper bound of the OPT algorithm

The study presented in [67] introduces novel methods for estimating the lower and upper bounds of the *OPT* algorithm, taking into consideration the actual file sizes. The paper presents two primary methods: *Flow-based Offline Optimal* (*FOO*) and *Practical FOO* (*PFOO*). *FOO* is noted for its high accuracy, but is computationally demanding, limiting its applicability to datasets with tens of millions of requests. In contrast, *PFOO* is designed for scalability, handling hundreds of millions of requests and providing nearly tight upper and lower bounds for *OPT*.

Prior to this work, the only known lower bound was the Infinite Cache model, represented as a horizontal line in *MRC* plots, leading to misconceptions about the potential for improvement.

This paper, in contrast, provides compelling proof that *PFOO-L* is a genuine lower bound of *OPT*. The algorithm operates under the premise that cache resources are finite in both space and time, introducing a “cost” metric for storing a file in the cache post access, calculated as the product of reuse distance and object size. In this model, each file request is treated as an interval until the file is requested again. *PFOO-L* then adopts a greedy approach, selecting the smallest intervals until the cache size limit is reached. Each selected interval equates to one cache hit, although these intervals may overlap, causing the aggregate size of stored files to exceed the cache limit. This overlap renders *PFOO-L* a lower bound. However, if file requests are independently distributed, this overlap is not significantly detrimental. Thus, *PFOO-L* establishes itself as a lower bound for *FMR*, as no other caching strategy can achieve fewer file misses with the same total resources ($N \times C$).

3.4.3 Lower Bound for BMR of Reactive Policies

The *PFOO-L* algorithm, as an approximation of *OPT*, effectively optimises the number of cache hits and misses. However, optimising the volume of data associated with these hits and misses, specifically in bytes, is also crucial. To address this, a modified version of *PFOO-L* has been developed that can be employed to approximate a lower bound for the *OPT* algorithm when measuring performance in terms of byte volume, referred to as the *OPT.Bytes* algorithm. This adaptation focuses on the byte-level impact of caching decisions, providing a more comprehensive understanding of cache efficiency.

It has to be noted that N denotes the total number of requests in the trace. The trace contains M unique files $f_1, f_2, \ldots, f_M$ with sizes $s_1, s_2, \ldots, s_M$, respectively. The i-th request can be represented by the pair $\{i, f_{j_i}\}$, where j_i is the identifier of the requested file. Let $T_{dif}[i]$ denote the reuse distance, i.e., the difference between the order of the future request for the same file and the current request. C denotes the cache capacity in bytes.

Similarly to [67], the total cache resource is represented by an initially empty rectangle with sizes N and C, the resources are limited in time and space.

With each request $\{i, f_{j_i}\}$ a rectangle with height s_{j_i} and width $T_{dif}[i]$ can be associated and be placed between i and $i + T_{dif}[i]$ on the time axis. Its area $s_{j_i} \times T_{dif}[i]$ corresponds to the total amount of cache resources that should be allocated to file j_i to avoid the following request, at time $i + T_{dif}[i]$, to produce a miss.

The *PFOO-L* algorithm greedily picks the rectangles with the smallest area until all cache resources are consumed, and the sum of the areas of the placed rectangles exceeds the global rectangle size, regardless of the overlaps.

It therefore finds a lower bound for *FMR* of reactive policies, since no other reactive caching algorithm can get fewer misses using $N \times C$ total resources. In particular, the rectangles selected by *PFOO-L* may overlap in such a way that the required instantaneous capacity, the sum of the rectangles' heights, exceeds the constraint C.

Now it can be described how to adapt *PFOO-L* to find a lower bound for the minimum *BMR* (*PFOO-L.Bytes*). While in *PFOO-L* every selected rectangle brings a gain equal to 1 as it prevents a miss, in *PFOO-L.Bytes*, the rectangle corresponding to the request $\{i, f_{j_i}\}$ has an associated gain of s_{j_i}, i.e., equal to the bytes it prevents from downloading. This leads to a knapsack problem, where each file is associated with a cost to store it and a potential gain. A lower bound for *BMR* can be found by greedily selecting the rectangles with the best gain/cost ratio, until exhausting the caching resources. This leads to a knapsack problem.

The algorithm is:

1. Sort the intervals in the non-increasing order of *P/S*, where profit in this case is the file size, and cost corresponds to *FileSize* $\times$ *TimeTillNextRequest*, which is the same as 1/*TimeTillNextRequest*.

2. Greedily add intervals to the cache starting with the smallest until the next interval is too big to fit.

3. The solution would be the intervals that are already in the cache, plus a fraction of the first interval that does not fit.

By adding this fraction, the optimal solution of the knapsack problem is either matched or exceeded. Therefore, it's an upper bound for the hit bytes ratio and a lower bound for the miss bytes ratio.

Knapsack Problem Fractional Greedy Approximation

Problem statement. Assuming that there is a set of N items, each with profit p_i and size s_i, and a knapsack of size S. The task is to find a subset of items $I \subset [N]$ that maximises $\sum_{i \in I} p_i$ subject to the constraint $\sum_{i \in I} p_i \leq S$.

Greedy Fractional Approximation. The approximation is constructed in three steps:

1. Sort the elements in the non-increasing order of $\frac{p_i}{s_i}$. The new order of the elements is $i_j, j = \overline{1, N}$.

2. Greedily put items into the cache until $j == k$ so that

$$(\sum_{j=1}^{k} s_{i_j} \leq S) \wedge ((k == N) \vee (\sum_{j=1}^{k+1} s_{i_j} > S)). \tag{3.1}$$

3. If $k == N$ or $\sum_{j=1}^{k} s_{i_j} == S$, then this solution is optimal. If not, the left space $S - \sum_{j=1}^{k} s_{i_j}$ will be filled with a corresponding fraction of the $k + 1$ element. The final profit is:

$$\sum_{j=1}^{k} p_{i_j} + \frac{s - \sum_{j=1}^{k} s_{i_j}}{s_{i_{k+1}}} \times p_{i_{k+1}}. \tag{3.2}$$

Explanation of the solution. Since the elements with the highest $\frac{p_i}{s_i}$ ratio have been taken, it is impossible to obtain a higher profit with a total space of S . Therefore, this solution gives a profit which is either equal to the optimal, or exceeds it. In case $\sum_{j=1}^{k} s_{i_j} == S$, the solution is optimal.

Continuation of the Approach: Detailed Implementation of *PFOO-L.Bytes*

The complete algorithm *PFOO-L.Bytes* is presented in Algorithm 1. Arrays R, T and S are initialised with the request sequence, the order of requests, or request time, and the sizes of the request files, correspondingly. The first step is to find the order of the next request $T_{next}[i]$ for the same file f_{j_i} for each request $\{i, f_{j_i}\}$ of the trace (line 7). If the file f_{j_i} is accessed for the last time, $T_{next}[i] = \infty$. Next, the reuse distance between consecutive requests $T_{dif}[i]$ to the same file f_{j_i} is calculated (line 8). This leads to the rectangles' sizes (line 9), which are sorted by density, i.e., the gain/cost ratio (line 10), and added to the cache while there are enough caching resources (lines 11–17). Since Algorithm 1 is not an actual caching algorithm, but only finds a lower bound of the optimal performance, the caching of file fractions is allowed. In the function `cache_file`, the second argument indicates the fraction of the file that needs to be cached. The final step is to add a fraction of the first rectangle that did not fit (lines 18–20). Note how implicitly solving the fractional knapsack problem is being solved, and then the optimal solution of the original knapsack problem is either matched or exceeded.

Algorithm 1: *PFOO-L.Bytes*

```
1  $R = [], T = [], S = []$
2  for $i = 1$ to $N$ do
3      append(R, $f_{j_i}$)
4      append(T, $i$)
5      append(S, $s_{j_i}$)
6  end
7  $T_{next} \leftarrow$ find_next_access(R)          // Define the next access time
8  $T_{dif} = T_{next} - T$
9  $I = S \times T_{dif}$
10 $I \leftarrow$ sort_by_density(I, S)
11 $i = 1$
12 $P = N \times C$
13 while $i \leq N$ and $I[i] \leq P$ do
14     cache_file($i, 1$)
15     $P = P - I[i]$
16     $i = i + 1$
17 end
18 if $i \leq N$ then
19     cache_file($i, P/I[i]$)
20 end
```

Both *PFOO-L* and *PFOO-L.Bytes* can be constructed through a single pass over the trace for different cache sizes. In this case, the preprocessing steps, finding the time of the next access and sorting the intervals accordingly, take $O(N \log N)$ time and $O(N)$ space, and iterating over the trace takes $O(N)$ both in time and space.

3.5 Exploring Enhanced *LRU* Variants: Implementation of *2-LRU* Caching

To extend the focus beyond the traditional *LRU* caching policy to include its variant, *2-LRU*, which has demonstrated superior performance in numerous scenarios, as outlined in previous studies [38, 76]. The effectiveness of both *LRU* and *2-LRU* is particularly notable in scenarios with high temporal locality, as evidenced in the access trace depicted in Figure 2.8.

The *2-LRU* algorithm, at its core, is designed to leverage the benefits of a two-layer caching system. This approach was considered due to its potential to enhance cache hit rates and overall efficiency. In the *2-LRU* model, the first layer, denoted as $l1$, operates as a simulation layer, where only file keys are stored without the actual file contents. This layer employs an *LRU* eviction policy, considering the sizes of the files. The second layer, $l2$, functions as the actual physical cache. Files are transferred to this layer only if they register a hit in the first layer, ensuring that frequently accessed files are prioritised. The eviction policy for the second layer is also based on the *LRU* principle.

This analysis specifically considers the case where both the $l1$ and $l2$ layers of the cache are configured to have the same size, offering a balanced and uniform structure for evaluating the *2-LRU* algorithm's performance. The policy is then presented in Algorithm 2.

The *2-LRU* algorithm generally exhibits superior performance compared to the traditional *LRU*, particularly due to its enhanced handling of files with frequent accesses. While it is notable that files accessed twice in a short interval, but not a third time are excluded from the cache in the *2-LRU* system, the overall advantages of *2-LRU* in efficiently managing cache hits typically outweigh this limitation, making it a more effective caching strategy in various scenarios.

3.6 Caching Algorithms Taking into Account Specifics of WLCG Workloads

In addition, new caching policies that take advantage of some specifics of WLCG workloads have been proposed and evaluated. In particular, these policies rely on the dataset membership information, which was obtained for each file using an additional data source, the Rucio metadata. Given that datasets are typically accessed in their entirety, if some files in the dataset are currently in use, one may expect that the other files within the same dataset will be accessed in the near future. In particular, the following policies

Algorithm 2: *2-LRU*

```
1  File $f$ is requested
2  if $f \in l1$ and $f \in l2$ then
3      $f$ is served (it's a hit)
4      $f$ is moved to the heads of both $l1$ and $l2$
5  end
6  else if $f \in l1$ and $f \notin l2$ then
7      it's a miss
8      $f$ is put into $l2$ (the head)
9      ensure the cache capacity of $l2$
10     $f$ is moved to the head of $l1$
11 end
12 else if $f \notin l1$ and $f \notin l2$ then
13     it's a miss
14     $f$ is put into $l1$ (the head)
15     ensure the cache capacity of $l1$
16     if $l2$ has enough free space for $f$ then
17         $f$ is put into $l2$ (the head)
18     end
19     else
20         nothing happens in $l2$
21     end
       // In this case, some files might be evicted from $l1$, but stay
          in $l2$, which leads to the situation:
22 end
23 if $f \notin l1$ and $f \in l2$ then
24     $f$ is served from $l2$ (it's a hit)
25     $f$ is moved to the head of $l2$
26     $f$ is inserted into $l1$
27     ensure the cache capacity of $l1$
28 end
```

are proposed, which preserve *LRU's* low complexity and are in addition particularly suited to serve high-rate request processes.

For several operational reasons it cannot be assumed that datasets are always processed in their entirety, or in a defined sequence:

- **Job Interruptions**: Jobs processing these files might face unexpected interruptions or failures, leading to a break in the sequence of file processing.
- **Preliminary Testing on Subsets**: Before processing the entire dataset, jobs are often initially tested on a smaller subset. This preliminary run can introduce variances in the order of file processing.
- **Concurrent Access by Multiple Users**: The same dataset might be accessed concurrently by different users, each running jobs that process the files. This simultaneous access can disrupt the traditional sequence.
- **Parallel Processing by Rucio**: Rucio, the data management system, reduces time to completion of tasks by running jobs in parallel. However, this parallel processing means, that different jobs might begin processing on different segments of the dataset, further deviating from an expected sequence.

In light of these operational intricacies, while the inherent sequence of files in dataset lists exists, the actual order of processing can vary considerably based on multiple factors.

Dataset Evict LRU/MRU. These policies insert a file into the cache only upon a miss for that file (as *LRU* does). They maintain information about the last access to a dataset and, when space is needed, they start evicting files belonging to the least recently accessed dataset. Among the files within this dataset, *Dataset Evict LRU* first evicts the least recently accessed files, while *Dataset Evict MRU* evicts the most recently accessed ones. It has to be stressed that both policies operate on the file level.

Dataset LRU. This policy relies on prefetching, as, upon a file miss, all files belonging to the accessed dataset are retrieved from the remote server and stored in the cache. Similarly, when cache space is needed, all files of the least recently accessed dataset are evicted. In short, this policy operates on individual files, it has to be noted that datasets are only logical concepts, unknown to the underlying file system, but practically behaves as *LRU* would if datasets were the atomic cacheable units.

A variant of *Dataset LRU* that is less aggressive upon eviction has also been tested. This policy does not evict entirely the least recently accessed dataset, but only as many of its files as needed. While this variant better uses the available storage space and achieves smaller *FMR* and *BMR*, the improvement is negligible, in the order of 10^{-4}.

3.7 Performance Comparison

Simulation of the cache behaviour using the same three-month trace reflecting the request process allowed for the evaluation of the performance of the caching policies. In each case, the initial state of the system is an empty cache.

Figures 3.3a and 3.3b compare how *FMR* and *BMR* change depending on the cache size, which is measured as a percentage of the total volume of unique files seen in the trace. *PFOO-L* and *PFOO-L.Bytes* algorithms provide lower bounds for the performance of any reactive caching policy. *Dataset Evict LRU* and *Dataset Evict MRU* show a negligible difference in performance of the order of 10^{-3}, so results are only shown for the first of them.

It is essential to ascertain the optimal sampling fraction of total storage that yields significant benefits in caching performance. Empirical evidence suggests that utilising merely 20-50% of the total storage capacity can achieve a substantially high hit ratio, indicating the efficiency of this scaled storage approach in cache management.

Moreover, these plots show that, among the considered reactive techniques considered, *LRU* remains the best option. Differently from what was observed in many previous studies [38, 76], *2-LRU* results in significantly worse *BMR* and almost the same *FMR*.

A possible factor contributing to it could be the number of single requests. All the files accessed only once should be better for *2-LRU* (since it will never put them into the cache and will not occupy the space).

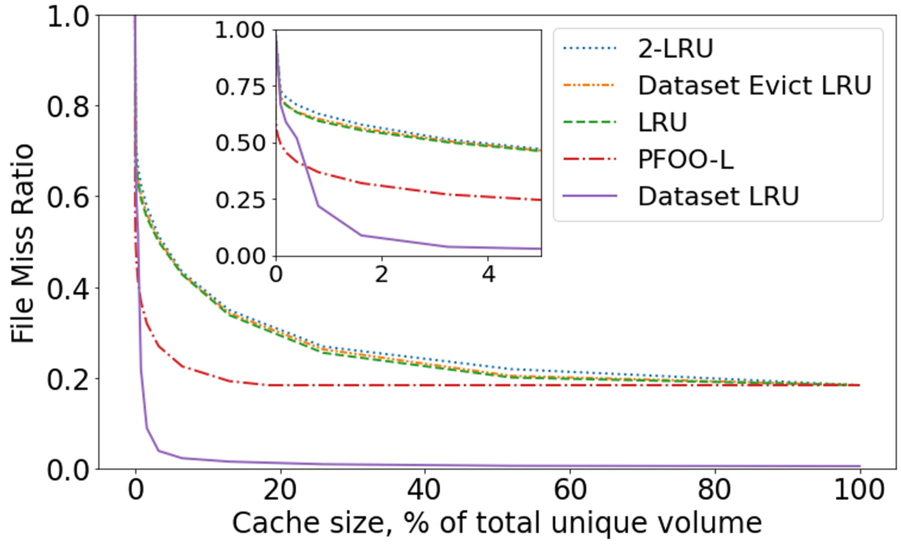
Curves for *LRU* and *Dataset Evict LRU* almost coincide. As most of the files in the same dataset are accessed consecutively, files in the least recently accessed dataset are to a large extent also the least recently accessed files. Cache states are then almost identical for *LRU* and *Dataset Evict LRU*.

As expected, all reactive techniques (*LRU*, *2-LRU*, *Dataset Evict LRU*, *PFOO-L*, *PFOO-L.Bytes*) provide the same performance when the cache can store the whole catalogue. In these cases, misses occur only the first time a file is requested (cold misses) and are not due to space constraints.

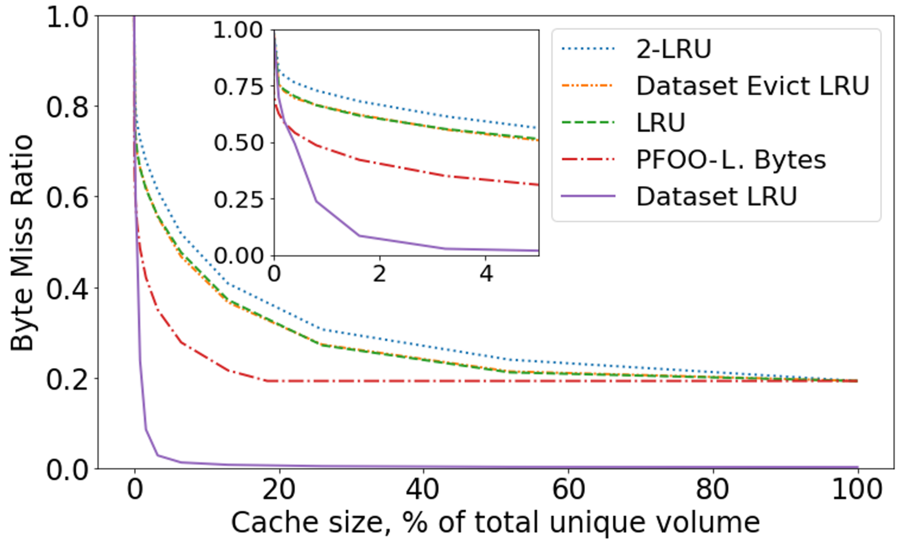
A direct comparison between *Dataset LRU* and the rest of the algorithms is challenging due to the prefetching this algorithm incorporates, which alters the number of cold misses.

This difference persists even in scenarios where the cache size equals 100% of the total data volume. To facilitate a more accurate comparison, cold misses have been excluded from the analysis, focusing solely on the shape of the resultant curves, illustrated in Figure 3.4.

Under these modified conditions, the *Dataset LRU* algorithm continues to outperform



(a) *FMR* as a function of cache size.



(b) *BMR* as a function of cache size.

Figure 3.3: Detailed comparison of various cache eviction policies' performance in terms of *FMR* and *BMR* depending on the cache size. The plots evaluate the effectiveness of different cache eviction policies from January to March 2020 for the ATLAS experiment, by showing their *FMR* and *BMR* as the cache size varies from 0% to 100% of total unique volume. Each line represents the trend in miss ratios for a different policy. The insets provide a zoomed-in view of the *FMR* and *BMR* for cache sizes ranging from 2% to 4%.

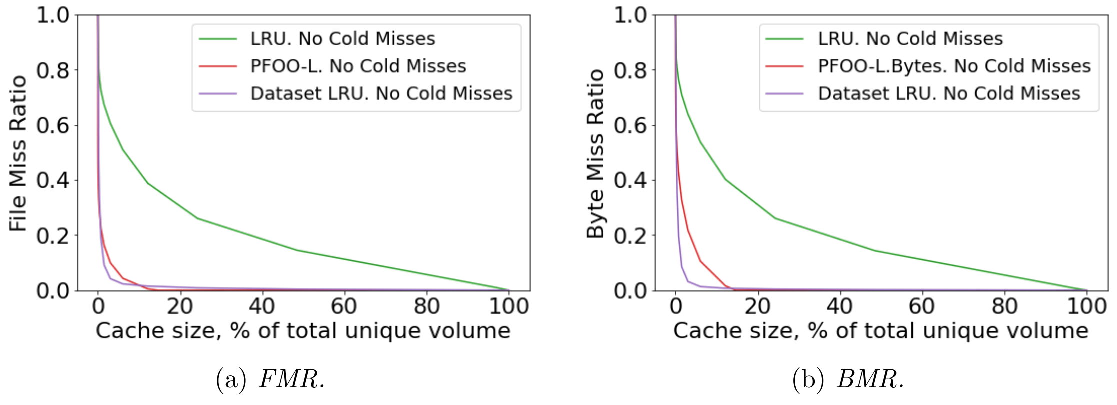


Figure 3.4: Comparative analysis of FMR and BMR excluding cold misses for various cache eviction policies. The plots evaluate the effectiveness of different cache eviction policies from January to March 2020 for the ATLAS experiment, by showing their *FMR* and *BMR* as the cache size varies from 0% to 100% of total unique volume. Each line represents the trend in miss ratios for a different policy. Cold misses, which occur when data requested has never been in the cache, are excluded from this analysis.

the optimal lower bound. This outcome is expected, considering the impact of prefetching on enhancing cache hit rates.

3.7.1 Implemented Enhancement Strategies

Since the dataset size is so diverse (Figure 2.20) both in terms of the number of files and the number of bytes, sometimes, when running *Dataset LRU*, evicting several files instead of evicting the whole dataset would be sufficient to make space for the newly requested dataset. Therefore, there is possibly an opportunity to improve the algorithm by switching to a file-based eviction, while still prefetching whole datasets. To estimate the influence, the simulations have been run with a buffer space equal to the average size of the dataset, while maintaining the eviction rule. In this way, the dataset is only evicted once there is a need for most of the space occupied by it. The difference in the result is in the order of 10^{-4}, which is negligible.

Another possible improvement would be to account for the partial hits while a dataset is still in the loading queue. In the implementation, a pessimistic scenario is run, where all the requests for a dataset result in a miss until the dataset is fully loaded into the cache. A less conservative set-up would be to account for them as partial hits, proportionally to the part of the dataset already loaded. Again, there was a very slight improvement seen; the difference is in the order of 10^{-3}.

3.8 Limited Connectivity Throughput Study

So far, it has been assumed that the throughput between the processing grid site and the data source is large enough to make the content retrieval time negligible in comparison to the inter-request arrival times. In reality, as described in Sec. 1.4.3, limited connectivity could lead to delayed hits and significantly influence the cache performance. To estimate this effect, an array of experiments have been conducted that emulate the retrieval process under throughput constraints and compare the performance of *LRU*, as the best reactive policy, and *Dataset LRU*, which showed the best performance previously.

3.8.1 Implementation Specifics

The backhaul link and the main storage are modeled as a single-server *FIFO* (*First In First Out*) queue [77] with a constant service rate, corresponding to the throughput. The customers in the queue are the objects to be retrieved, a file in the case of *LRU*, a dataset in the case of *Dataset LRU*. Such a queue is referred to as the "loading queue." The order of the objects in the queue represents the order in which they are to be loaded into the cache. Objects are inserted into the cache when their service is completed. Each object request[3] is considered as one iteration of the algorithm. The following explains how the algorithm works.

When an object is accessed for the first time, it is put into the loading queue. The objects in this queue are served in the *FIFO* manner. If there is already a pending request for that object, the new request will not influence the priority, and the previous loading order will be maintained.

Simultaneously with this, a second data structure serves to maintain the cache order according to the caching policy. The head of the cache corresponds to the *MRU* objects and the tail to the *LRU* ones. Upon each object request, it is inserted into the *MRU* place in the cache, either inserted as a new element or moved to the head if it was already in the cache. Initially, a `not_loaded` tag is attached and changed to the opposite when the object is loaded fully. In contrast to the loading queue, the object order in the cache changes with each new access. Consequently, the tail of the cache contains the next candidate for the eviction. If the accessed object was present in the cache with a tag `loaded`, it is also counted as a hit before being moved to the head of the cache.

Before each new object request, a simulation of the loading process is performed. This is done by calculating how many bytes could have been loaded into the cache, given the bandwidth and the time difference with the previous access. Then, the algorithm

[3]each file access record in the generated trace

iterates over the loading queue, starting with the head, and distributes the loaded bytes amongst the objects that were the first in the queue. The fully loaded files are then put into the cache, their tag changes to `loaded`, where the *LRU* order will be maintained.

With each iteration, the occupied size of the cache is updated. When the volume of the loaded bytes exceeds the maximum cache capacity, the algorithm starts removing objects from the cache in *LRU* order, i.e. starting from the tail. The only exception to the eviction of the *LRU* object is when this object is the first one in the loading queue, and the reason for the current cache cleaning is to make space for this exact object. In this case, this object is skipped and one but the last cache element will be removed. Since the first element of the queue is never removed and loading the next object does not start before the first one is accessed, remove partially loaded objects are never removed. Therefore, the possibility that the removed file was still in the loading queue is eliminated.

The Algorithm 3 portrays the implementation details in the form of the pseudocode of the above-mentioned algorithm for the case of file-based *LRU*. The bandwidth becomes one of the parameters of the system. The model is simplified to loading files into the cache one by one according to the request sequence, facilitating the evaluation of the algorithm's efficiency in handling file requests with bandwidth constraints.

In the case of *Dataset LRU*, the additional problem of fetching the whole datasets into the cache arises: some of them do not fit when simulating small-sized caches. In this case, simply skipping them from loading is not an option, since at least the requested file must be served to the user. This case therefore requires more granular cache insertion rules, based on files or datasets, depending on the size factor.

3.8.2 Observations on the Queue Length

When studying the queue sizes, it was noticed that it does not always oscillate around zero. At the beginning of the trace, the queue keeps growing while the cache is filling up, then its size either falls down to zero, or stabilises and starts oscillating around a non-zero value, or keeps growing. The last scenario has no clear interpretation, since some requests will never be served to the user. These cases as marked dashed lines on the follow-up plots (Figure 3.8b and 3.9b).

LRU: Figure 3.5 presents how the length of the loading queue is behaving depending on the connectivity. For 10 Gbit/s the situation is completely unstable . The queue keeps growing, which means the some requests will never be served. This implies that the network's bandwidth is insufficient to handle even the average request rate. For the 20 Gbit/s connectivity, the situation depends on the cache size. If the cache capacity is large enough to store 50%+ of the data, the queue either decreases in size (goes to 0)

Algorithm 3: *LRU* with delayed hits

```
Data: $cache\_cap$ - the maximum cache capacity
Data: $Fid = fid_1, fid_2, \ldots, fid_N$ - the sequence of requests
Data: $Fsize = fsize_1, fsize_2, \ldots, fsize_N$ - file sizes corresponding to the request
      trace
Data: $Ts = ts_1, ts_2, \ldots, ts_N$ - the timestamps of the requests
Data: $connect$ - the connectivity to the remote site
$Cache \leftarrow deque()$
$Queue \leftarrow deque()$
$cur\_cap = 0$
$prev\_ts = 0$
$hits = 0$
$i = 1$
while $i \leq N$ do
    $loaded = connect \times (Ts[i] - prev\_ts)$
    $prev\_ts = Ts[i]$
    while $!Queue.isEmpty()$ AND $loaded > 0$ do
        $Queue.load\_first(loaded)$
                                  // involves updating $loaded$ and $cur\_cap$
        while $cur\_cap > cache\_cap$ do
            $fid, fsize \leftarrow Cache.pop()$
            $cur\_cap = cur\_cap - fsize$
        end
        if $Queue.first\_is\_loaded()$ then
            $Cache.put(Queue.pop())$
        end
    end
    if $Cache.contains(Fid[i])$ then
        $Cache.move\_to\_head(Fid[i])$
        $hits = hits + 1$
    end
    else
        if $!Queue.contains(Fid[i])$ then
            $Queue.put(Fid[i], Fsize[i])$
        end
    end
    $i = i + 1$
end
```

or stabilises and becomes of a constant size with some oscillations. If the cache size is smaller than that though, the queue keeps growing. This situation arises because while the bandwidth can accommodate the "average" request rate, it falls short during peak requests. Consequently, a buffer in the form of adequate cache size is necessary to enable the system to even out the request load. 50 Gbit/s connectivity seems to be sufficient to deal with the request rate and with the peaks. These peaks could be easily observed on the next plot (for 100 Gbit/s connectivity) where there are clusters of requests. Such peaks correspond to clusters of misses which require the retrieval of large volumes of data. With a 100 Gbit/s connectivity, the queue size quickly goes does to zero, which does not imply any additional misses. For such a level of connectivity, the network is almost never congested: the queue occupancy is close to zero most of the time, with some peaks appearing only for small cache sizes (3% and less) or ($\leq 6\%$).

In conclusion, with at least 50 Gbit/s, the system remains stable, even with small cache sizes of 2% of the total volume, but some time is required to load the whole queue again.

Dataset LRU: The analogous plots are presented in Figure 3.6. For 10 Gbit/s, the system is unstable, the queue keeps growing. It can be seen that after 3e7 requests the queue size decreases, but looking at the previous plot, it can be seen that simply there are not many requests coming in this period of time, so the system manages to process some part of the queue. 20 Gbit/s is enough only if the cache size is 52%+ of the total volume. If not, the system is unstable, the queue keeps growing. For 50 Gbit/s connectivity, 13%+ is sufficient for the system for stable operation . In the other cases, the queue length never goes down to zero, at least on this trace, so it's hard to argue that the system is stable. 100 Gbit/s connectivity - with 3%+ of the total size, the queue length goes down to 0. For most of the period and most of the cache sizes, it stays close to zero, which means the throughput is sufficient to keep up with the incoming requests.

3.8.3 Full-Cache Points and the Calculation of Hit Ratios

The point of the full cache is the point when the cache is fully populated for the first time. It is obvious that, for any reasonable cache strategy, everything that happens before this point does not exhibit the behaviour of the cache eviction policy, since it has before this point not been applied. Comparing the total hit count and hit byte numbers of these simulations would not be representative enough of the eviction policy performance in the long run. Plus, comparing them when the bandwidth connection is different is not correct since the point when the caches are fully populated is not the same in these cases.

An alternative approach of simply discarding cold misses prior to the cache reaching

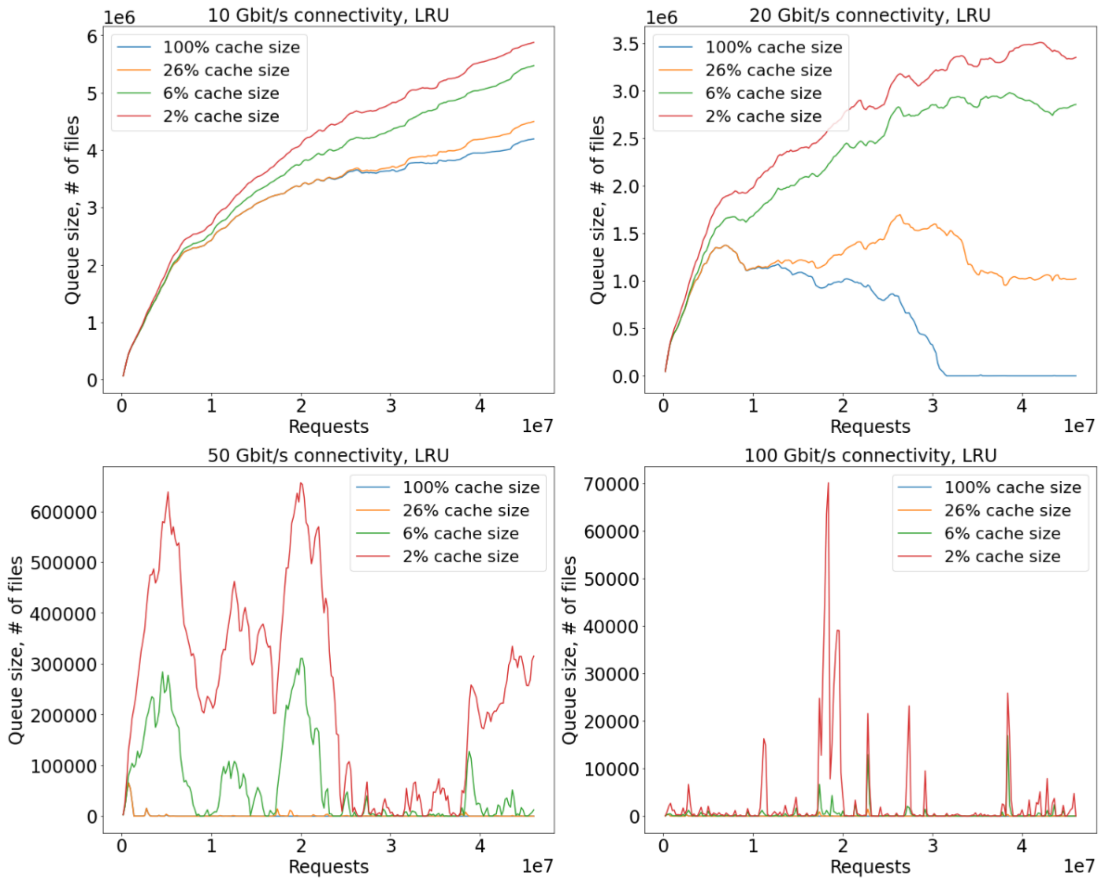


Figure 3.5: Queue size variation over time for *LRU* cache policy under different network throughputs. This figure illustrates how the queue size, expressed as the number of files, evolves over time with respect to the number of requests for different cache sizes under 10 Gbit/s, 20 Gbit/s, 50 Gbit/s, and 100 Gbit/s connectivity scenarios using *LRU* cache policy. Data encompasses 46 million requests, covering the entire three-month period from January to March 2020 for the ATLAS experiment, across cache sizes of 100%, 26%, 6%, and 2%.

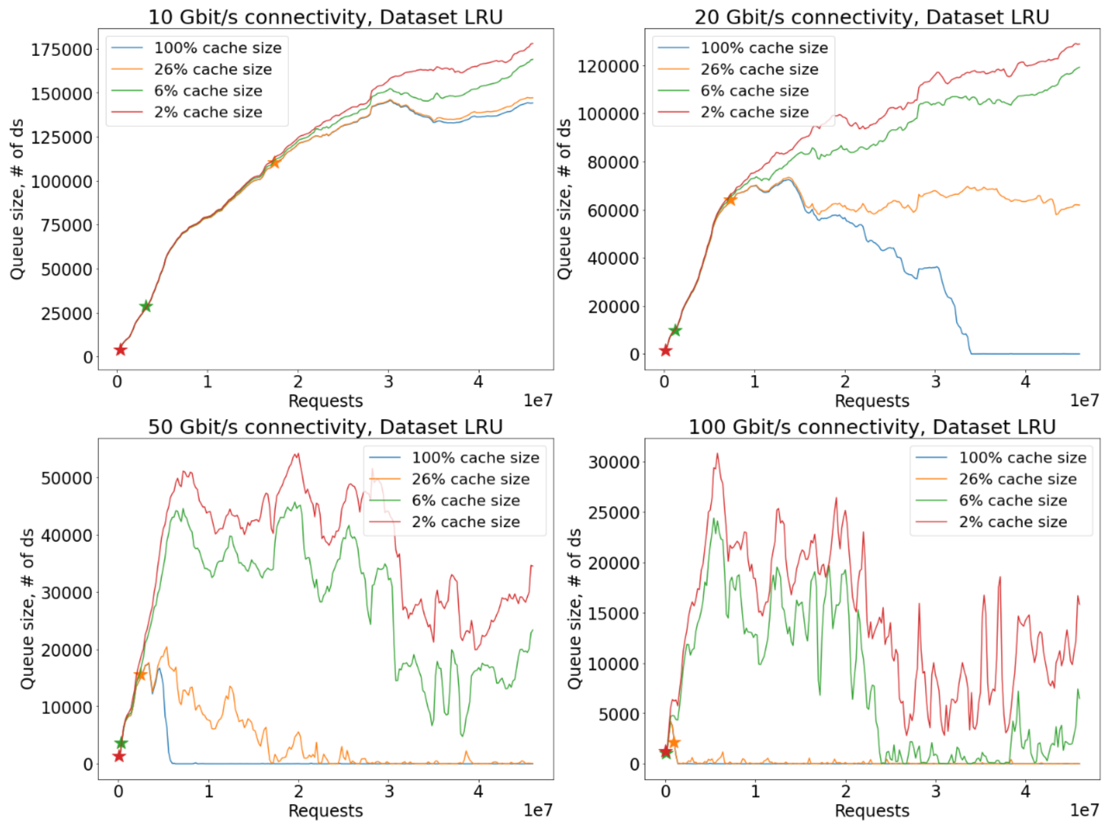


Figure 3.6: Queue size variation over time for *Dataset LRU* cache policy under different network throughputs. This figure illustrates how the queue size, expressed as the number of files, evolves over time with respect to the number of requests for different cache sizes under 10 Gbit/s, 20 Gbit/s, 50 Gbit/s, and 100 Gbit/s connectivity scenarios using *Dataset LRU* cache policy. Data encompasses 46 million requests, covering the entire three-month period from January to March 2020 for the ATLAS experiment, across cache sizes of 100%, 26%, 6%, and 2%. The star symbols mark the initial points when the cache reaches full capacity.

its full capacity is also impractical. This is due to the likelihood that the cache could become full towards the end of the request sequence, especially if even a single new file is accessed, which is a common occurrence. Moreover, when comparing different cache eviction policies, it is important to note that the comparative analysis is conducted along the y-axis of *MRCs*. In this dimension, all policies are equally affected by the same exact number of cold misses, thereby ensuring a consistent basis for comparison.

Another observation is that the lower bandwidth requires more time to reach the same set-up in terms of the hit rate, because in these cases the cache state is updated later.

Instead of calculating the hit ratio from the point where the cache is full for each set-up, one can start from the point (approximately, 40%) where it is known for sure that all the cache sizes with any bandwidth connection would be full. Otherwise, one compares the algorithms over different traces. For the full size of the cache, this is an exception, the cache will not be full yet, but to satisfy this condition, it would be required to cut off almost the whole trace. In this set-up, the new content is constantly arriving.

3.8.4 Discussion of Experimental Results

Figure 3.7 shows how the hit rate changes over time for different cache sizes. Lower throughput requires longer times to reach the same set-up in terms of the hit rate, since the cache state is updated slower. The squares on the curve designate the first time instant when the cache is full. Lower throughput and larger cache sizes shift these points further to the right. The troughs observed during these simulations are caused by a combination of the peaks in the request rate observed in Figure 2.17 and the short time difference between consecutive accesses (Figure 2.9).

In order to eliminate the effect of the initial transient, the first 40% of the trace is considered as a warm-up period and evaluate the metrics of interests only on the final 60%. In this manner, it can be guaranteed that the cache is full for all the considered set-ups, like different throughputs, cache sizes, caching policies, but for the cache sizes closely approaching 100%.

The effect of throughput constraints on *LRU* performance is illustrated in Figure 3.8b. At a bandwidth of 100 Gbit/s, which is currently available between T0 and almost all T1 sites, the *Byte Hit Ratio* is almost identical to the *LRU*, which is the same when assuming the bandwidth is infinite. It becomes clear that as far as the system provides a throughput of at least 50 Gbit/s, delayed hits have little effect on *BMR*, which is very close to the ideal case of infinite throughput. At the same time, a throughput of 30 Gbit/s shows a significant difference in performance with small cache sizes (20% and

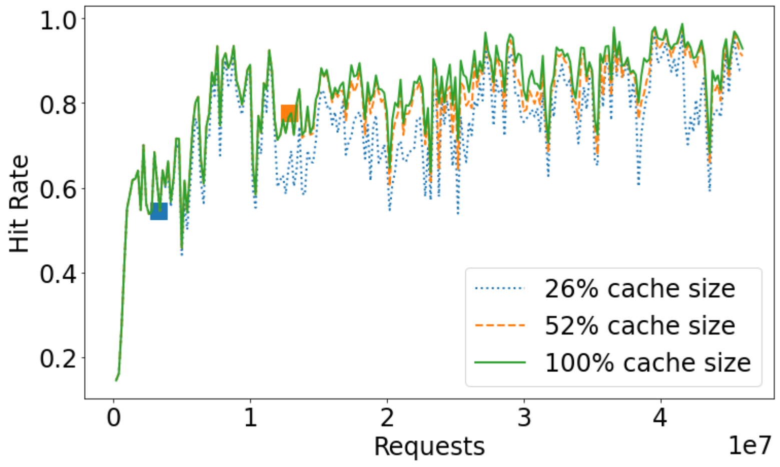


Figure 3.7: Hit rate over time for a 100 Gbit/s throughput using *LRU*, measured as the number of hits per request. The squares mark the initial points when the cache reaches full capacity. Data encompasses 46 million requests, covering the entire three-month period from January to March 2020 for the ATLAS experiment, across cache sizes of 26%, 52%, and 100%.

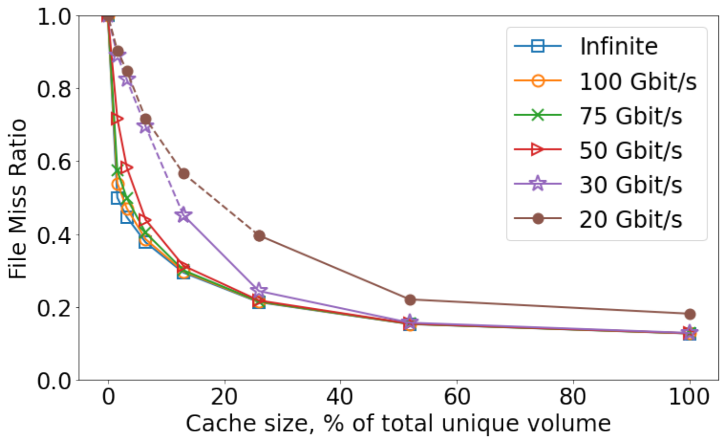
less). For 20 Gbit/s the *BMR* is higher for all cache sizes.

The situation with the *FMR* is very similar (Figure 3.8a). The results differ in absolute numbers, but the relative position of the *MRCs* is the same.

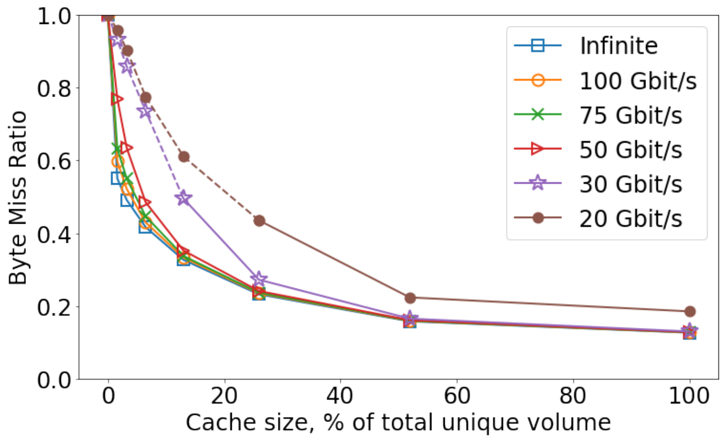
As the throughput decreases, scenarios arise where the rate of content retrieval requests consistently exceeds the connectivity throughput. This leads to an increasing queue occupancy over time, indicating a bottleneck where incoming data requests cannot be processed quickly enough due to limited bandwidth. In these pathological cases, the numerical values for *FMR* and *BMR* are not representative, since they heavily depend on the trace length, and are then represented through dashed lines in the *BMR* plots (Figure 3.8b and the following one for *Dataset LRU* 3.9b).

In contrast, the effect of the limited throughput on the performance of *Dataset LRU*—the best policy in the infinite throughput regime (Figure 3.3b)—is more profound (Figure 3.9b). In fact, upon a miss, *Dataset LRU* retrieves a much larger amount of data than *LRU* as the comparison of dataset sizes to file sizes shows (Figure 2.6a and 2.20). As the loading time increases, the delayed hits counteract the prefetching effect.

A 13% cache size, even at 100 Gbit/s throughput, which had virtually no influence on *LRU* performance, leads *Dataset LRU* to experience *BMR* equal to 0.15 compared to just 0.01 for the infinite throughput case. *Dataset LRU* still outperforms *LRU* in this scenario, but the relative performance improvement is significantly reduced. For even smaller throughputs and/or smaller cache sizes, the effect of delayed hits diminishes

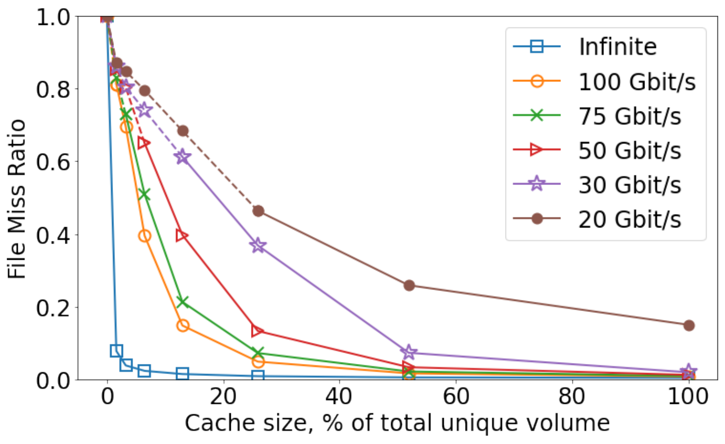


(a) *FMR* as a function of cache size.

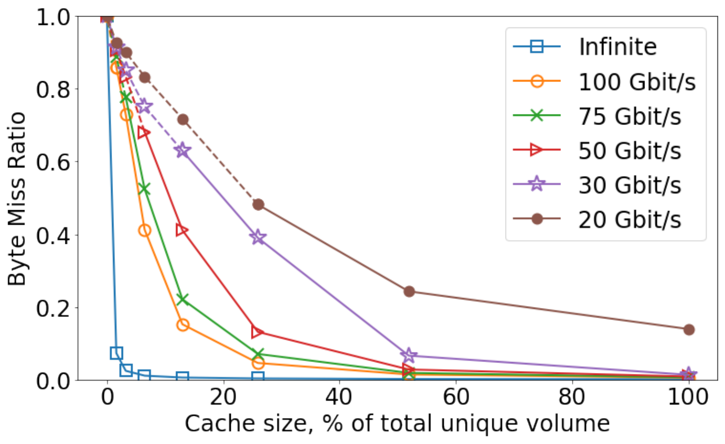


(b) *BMR* as a function of cache size.

Figure 3.8: Analysis of *LRU* cache performance across various throughput levels. This figure presents a detailed comparison of cache in terms of *FMR* and *BMR* under different network throughput conditions ranging from 20 Gbit/s to 100 Gbit/s, as well as an ideal scenario labelled “Infinite”. Panel (a) shows the *FMR* as a function of cache size, measured as a percentage of total unique volume. Panel (b) depicts the *BMR*. The analysis covers data calculated on the final 60% of the trace, with dashed lines representing scenarios where the loading queue size grows continuously.

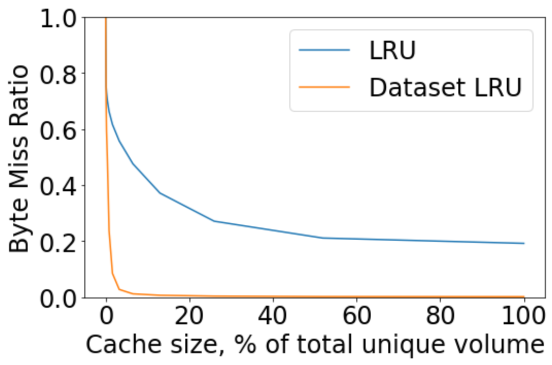


(a) *FMR* as a function of cache size.

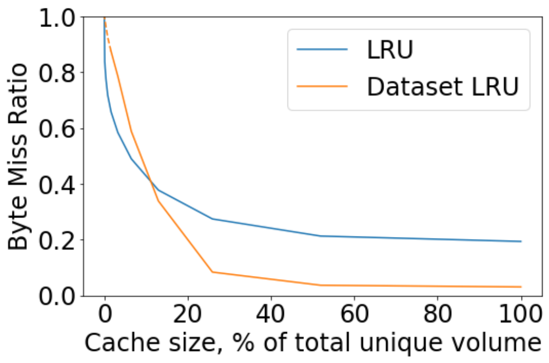


(b) *BMR* as a function of cache size.

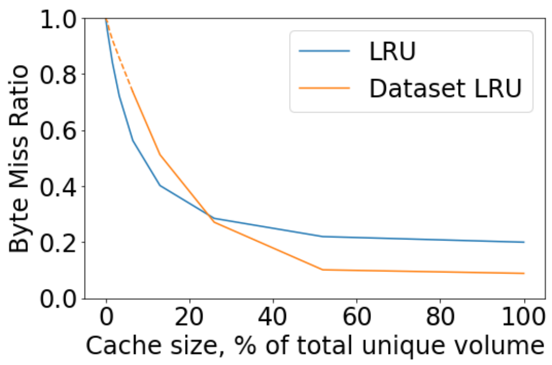
Figure 3.9: Analysis of *Dataset LRU* cache performance across various throughput levels. This figure presents a detailed comparison of cache in terms of *FMR* and *BMR* under different network throughput conditions ranging from 20 Gbit/s to 100 Gbit/s, as well as an ideal scenario labelled "Infinite". Panel (a) shows the *FMR* as a function of cache size, measured as a percentage of total unique volume. Panel (b) depicts the *BMR*. The analysis covers data calculated on the final 60% of the trace of the three-month trace Jan-Mar 2020 for the ATLAS experiment, with dashed lines representing scenarios where the loading queue size grows continuously.



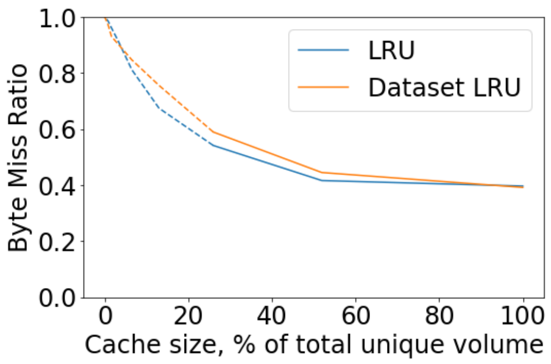
(a) Infinite throughput.



(b) throughput = 100 Gbit/s.



(c) throughput = 50 Gbit/s.



(d) throughput = 20 Gbit/s.

Figure 3.10: The plots compare the performance of *LRU* and *Dataset LRU* in terms of *BMR* under different throughput ("Infinite", 10 Gbit/s, 50 Gbit/s, and 20 Gbit/s). The analysis covers data calculated on the final 60% of the three-month trace Jan-Mar 2020 for the ATLAS experiment.

completely the advantage from prefetching, and Dataset *LRU* starts performing worse than *LRU*.

In order to compare the performance of the two algorithms, it is worthwhile to look at the intersection points of *BMRs* of *LRU* and *Dataset LRU*, depending on the throughput (Figure 3.10). *Dataset LRU* clearly does not always have an advantage over *LRU*, as in case of the infinite bandwidth (Figure 3.10a). With limited bandwidth, it takes more and more time to load the elements from the queue to put them into the cache, which in turn starts to level out the prefetching effect. With 100 Gbit/s connectivity, *Dataset LRU* starts overperforming *LRU* in terms of the *BMR* only with the cache capacity approx. 7% and more of the total unique volume. This intersection point shifts more and more to the left as the bandwidth decreases: Approx. 25% of the total volume for 50 Gbit/s connectivity (Figure 3.10c), and 90-95% for 20 Gbit/s (Figure 3.10d).

From these results, it becomes clear that the throughput must be taken into account when developing cache eviction policies for a remote computation model. The limited throughput worsens the performance of cache algorithms, not proportional to the results under the infinite throughput assumption. Figure 3.11 shows which caching policy should be preferred, between *LRU* and *Dataset LRU*, depending on the throughput and the cache size.

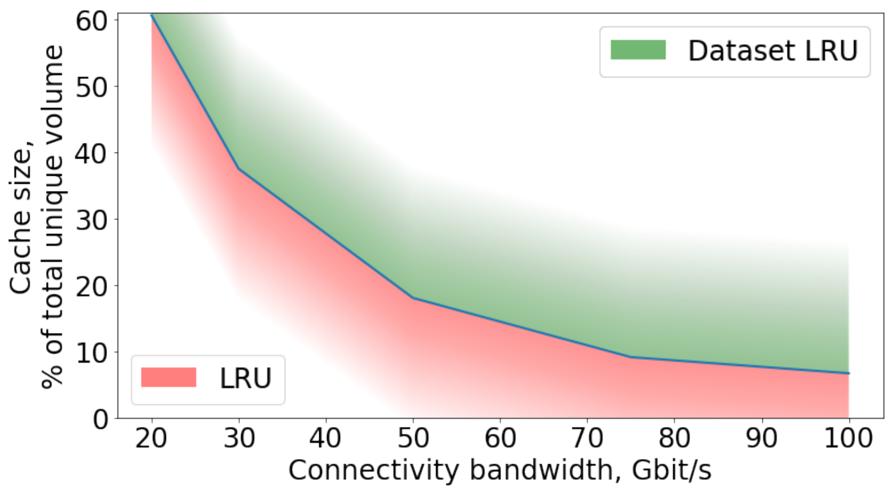


Figure 3.11: This figure illustrates the effectiveness of *LRU* and *Dataset LRU* caching policies across a range of connectivity bandwidths from 20 Gbit/s to 100 Gbit/s and varying cache sizes as a percentage of total unique volume. The shaded areas represent the efficiency range of each caching policy, with the red area denoting *LRU* and the green area representing *Dataset LRU*. As the connectivity bandwidth increases, the effective cache size required for *Dataset LRU* to outperform *LRU* decreases, indicating that *Dataset LRU* becomes more favourable at higher bandwidths. The analysis is based on the final 60% of the three-month trace Jan-Mar 2020 for the ATLAS experiment.

3.8.5 Prefetching Overhead

Prefetching data can help speed up data retrieval, but it also has its downsides. One significant drawback is that it can increase the total amount of data sent to a remote site compared to standard methods like *LRU*. This extra data might not be used, leading to inefficiency and potential resource waste. In essence, prefetching can result in an overhead, meaning that too much data is loaded in advance. This “prefetching overhead” can be visually understood through a comparison between the percentage of prefetching overhead and cache size. While the intention behind prefetching is to improve hit rates and reduce latency, unnecessary data retrieval can offset these benefits.

Furthermore, this approach resembles to a certain degree the current staging practice. Consequently, this direction wasn't pursued beyond the presented level. The reduction in miss ratio achieved through prefetching is offset by a significant increase in the accumulated data transfer volume, as depicted in Figure 3.12 and due to the granularity the impact on waiting jobs is also contra-productive. These trade-offs highlight the complexities involved in balancing caching efficiency with resource utilisation.

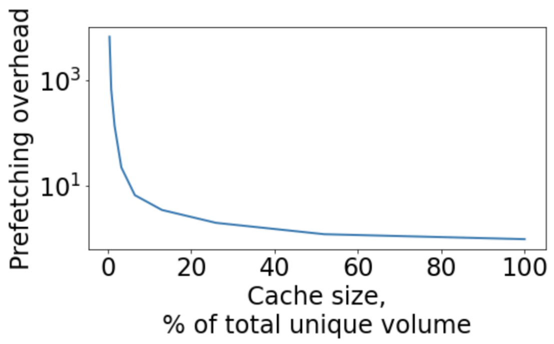


Figure 3.12: Prefetching overhead as a function of cache size. This graph illustrates the exponential decrease in prefetching overhead with increasing cache size, expressed as a percentage of the total unique volume. The overhead is measured by the ratio of accumulated transferred volumes for *Dataset LRU* compared to *LRU* under an infinite bandwidth scenario. This trend underscores the trade-offs between caching efficiency and resource utilisation. The analysis is based on the final 60% of the three-month trace Jan-Mar 2020 for the ATLAS experiment.

3.9 Bandwidth Exploration

The integrity and efficiency of data-driven operations, such as fetching individual files or prefetching entire datasets, hinge significantly on the quality of the connectivity to remote sites. As the backbone of these operations, a robust and consistent connection ensures the timely transfer and availability of data, preventing lags or "delayed hits" that might occur when data is not retrieved promptly. However, the challenge lies in discerning the optimal bandwidth necessary for these tasks. Factors such as the duration for which files remain open, the rate of data transfer, and the potential overlap in bandwidth demands from concurrent operations all play pivotal roles. Delving into this intricacy forms the basis for this investigation, trying to delineate the bandwidth prerequisites that will optimise these crucial processes.

Figure 3.13 presents the distribution of the cumulative bandwidth utilised during file openings, with each instance weighted according to the duration of its overlap. Essentially, this figure illustrates the distribution of the total reading bandwidth at the T0 site, in conjunction with the specific duration (measured in days) over a three-month period. It is observed that the total bandwidth rarely surpassed 100 Gbit/s, despite the system's capability to support higher reading speeds of up to 400 Gbit/s. Notably, the data reveals that for only 7.35% of the observed period, the total bandwidth exceeded 100 Gbit/s.

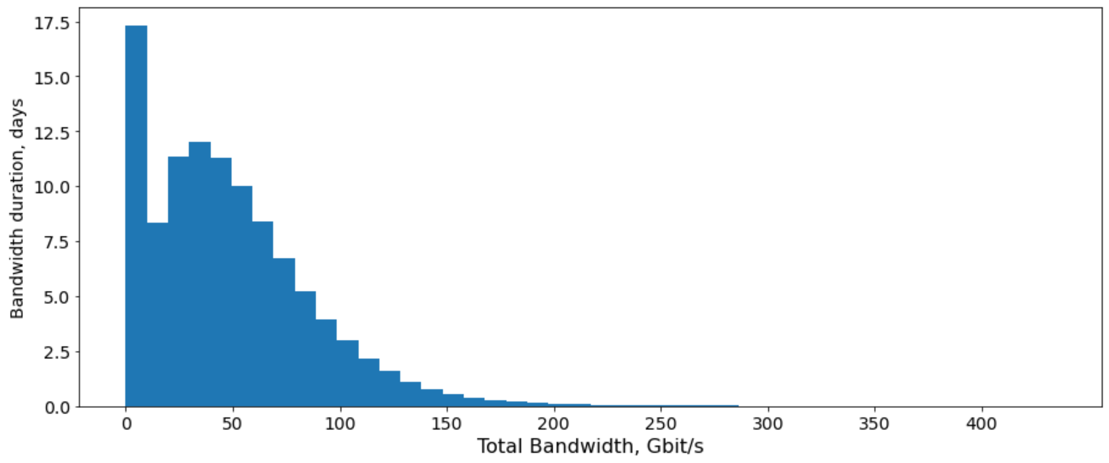


Figure 3.13: Distribution of cumulative bandwidth utilisation for file access events over time. This histogram illustrates the distribution of total bandwidth used during file opening events, measured in gigabits per second, and weighted by the duration of each event's overlap with others. The analysis is based on the three-month trace Jan-Mar 2020 for the ATLAS experiment.

Optimising Transfer Rates Optimising data transfer rates hinges on two primary aspects: first, reducing the total volume of data transferred, and second, ensuring that the available connectivity bandwidth isn't overwhelmed or saturated. Current observations, as illustrated in the previous bandwidth accumulation plots, suggest that bandwidth saturation isn't a pressing concern given the ample resources at the T0. Consequently, the immediate focus shifts to minimising transfer costs, specifically by reducing the cumulative amount of transferred bytes.

While prefetching is not inherently counterproductive in optimising transfers—there are scenarios where prefetching can balance reading bandwidth and stave off peak loads—it's essential to weigh its costs and benefits. Under the assumption that the bandwidth usage stays within the confines of the provided connectivity, the prefetching might inadvertently inflate transfer costs.

As a result, the current emphasis is on devising a reactive eviction policy that can outperform the standard *LRU* approach.

3.10 Conclusions

The research demonstrated the utility of constructing *Miss Ratio Curves* for understanding cache performance relative to cache size, particularly in heterogeneous file size environments.

A point-based method was employed, that strikes a balance between computational efficiency and the precision needed for representing diverse caching strategies.

LRU demonstrates commendable performance with favourable ease of implementation and maintenance. However, it raises the question of whether a better *BMR* can be achieved through the adoption of alternative cache eviction policies.

In an attempt to close this gap, cache eviction policies specifically tailored to the WLCG types of workloads have been developed, such as Dataset Evict *LRU* and *Dataset Evict MRU*, which leverage the observation that files within a dataset are very often accessed together during analysis jobs. Despite extensive exploration and efforts, it has not been possible to surpass the performance of *LRU* in previous attempts with *2-LRU* and *Dataset Evict LRU*.

Despite the absence of a reactive technique in this study that surpasses the performance of *LRU*, the comparative analysis with optimal bounds indicates substantial scope for improvement, hinting at the possibility of developing more efficient caching strategies in the future.

For example, for the 5–10% cache sizes, the miss ratios could be reduced by a factor of 1.5.

This study also highlighted how limited connectivity can significantly affect cache performance, necessitating the consideration of network throughput in cache design. While this is less severe for T0 and T1 located caches, it becomes more critical for some of the poorly connected T2s.

The overall findings emphasise the need for adaptable strategies that take into account file size variability, network conditions, and workload characteristics.

Chapter 4

Machine Learning-based Caching Policies

4.1 Description of the Trace:

In this chapter, the results of applying Machine Learning techniques to the problem of caching in the WLCG are presented. All the results presented here were based on a trace generated from the 2022 data accesses (February - April) for the ATLAS experiment. Main characteristics of the trace are presented in Table 4.1.

Table 4.1: Main characteristics of the updated 2022 file access trace. This table summarises various data access metrics recorded within the ATLAS Experiment's computing instance at the CERN T0 Data Center for the period Feb-Apr 2022.

Metric	**Value**
Number of Read Accesses	55,714,376
Read Volume	112.19 PB
Number of Files	11,864,276
Volume of Files	25.055 PB
Number of Datasets	285,469

4.2 Performance of the *Belady* algorithm

A theoretical policy capable of predicting future file accesses with certainty would be able to identify the optimal file for eviction at each step. When files have the same size, this policy is straightforward - it selects the file that will be accessed farthest in the future (known as *Belady* algorithm [64]). However, when file sizes are different, determining the best eviction strategy becomes more difficult and is an NP-hard [65] problem.

Here are the counterexamples that show the proof that *Belady* policy is not optimal in the case of heterogeneous file sizes.

- Suppose there are four files f_1, f_2, f_3, f_4 with the sizes $s_1 = 1, s_2 = 1, s_3 = 2, s_4 = 2$ respectively (measured in some units). The cache capacity is 5 and the trace is the following:

timestamp:	t_1	t_2	t_3	t_4	t_5	t_6	t_7
requested file:	f_1	f_2	f_3	f_4	f_3	f_1	f_2

  At time t_4, according to the *Belady* algorithm, both files f_1 and f_2 should be evicted, which results in 1 hit over the whole trace (for the file f_3). On the other hand, evicting the file f_3 instead would leave enough space for the new f_4 and result in 2 hits (for f_1 and f_2) in the end.

  Therefore, *Belady* is not optimal for maximising *FMR*.

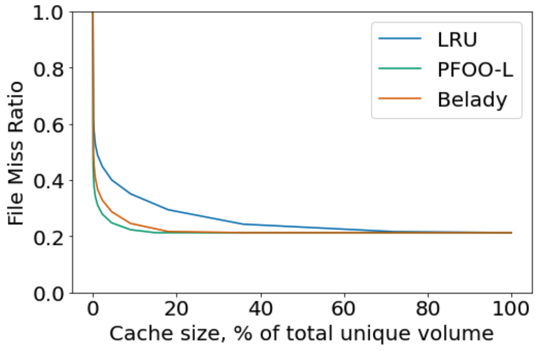
- Suppose the files f_1, f_2, f_3, f_4 have sizes $s_1 = 1, s_2 = 1, s_3 = 4, s_4 = 1$ respectively. The cache capacity is 6 and the trace is the following:

timestamp:	t_1	t_2	t_3	t_4	t_5	t_6	t_7
requested file:	f_1	f_2	f_3	f_4	f_1	f_2	f_3

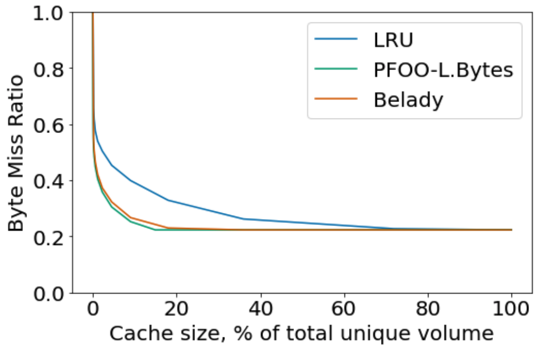
  When f_4 is inserted at time t_4, *Belady* evicts f_3, which is accessed the furthest in the future. This results in 2 hits and 2 hit bytes in total (at t_5 and t_6). Instead, evicting the file f_1 results in 2 hits and 5 hit bytes (at t_6 and t_7).

  Therefore, *Belady* algorithm is not optimal for maximising *BHR* either.

As seen previously in this analysis, the WLCG workloads are characterised by a vast spread in file sizes (reference Figure 2.12). Therefore, *Belady* algorithm would not be optimal for either *FMR* or *BMR* for the studied workloads. Nevertheless, the empirical observations present an unexpected outcome. When applying *Belady* algorithm to the extracted trace, it demonstrated performance that approached the theoretical lower bound for efficiency as defined by the *PFOO-L* and *PFOO-L.Bytes* algorithm, as illustrated in Figure 4.1a and Figure 4.1b. This suggests that for the analysed dataset, *Belady* algorithm exhibits nearly optimal performance.



(a) Comparing *Belady* and *LRU*. *FMR*.



(b) Comparing *Belady* and *LRU*. *BMR*.

Figure 4.1: Comparing the performance of *LRU*, *Belady* and the lower bound of the optimal algorithms using *MRCs*.

In practice, while not being optimal, *Belady* algorithm approximates an optimal lower bound for both *FMR* and *BMR*. This served as an inspiration to use *Belady* algorithm as a valuable example of nearly-optimal behaviours and combine it with the potential of Machine Learning tools for prediction.

4.3 Architecture of ML solution (2-Stage Approach)

It is important to remember that *Belady* algorithm, although nearly optimal on the extracted trace, cannot be practically implemented as it necessitates foreknowledge of the future, making it impossible for real-world implementations. Nonetheless, one of the feasible solutions could be to train a machine-learning model based on past cache accesses and employ it to guide future cache replacement choices. If the ensemble of features available in the trace of the previous accesses is a reliable indicator of future accesses, then the replacement strategy can approximate the performance of *Belady* algorithm.

Therefore, this task is separated into two stages: instead of training the ML model to answer the question “which file should be evicted from the cache” (like the paper [55] and possibly others), the approach here is to predict the reuse patterns on the files. Based on these predictions, a heuristic is then applied to determine eviction priorities. More specifically, the two stages are:

1. The initial stage involves building a robust predictor based on all the available data from the trace. This results in an ML model, which is trained with historical data to predict future file accesses.

2. The predicted access information is then utilised for cache eviction decisions.

Specifically, the predicted time (or probability) of file reuse is integrated into the caching model to make informed cache eviction choices and allows the cache to be freed up from files that are less likely to be accessed in the future.

4.4 Predicting Future Accesses

4.4.1 Search for Simpler Dependencies

Dependency Between File Lifetime and Access Rate

The initial step of this approach involved exploring simpler relationships between various features of the trace data, such as file size, distance to previous access, and the target variable, which is the reuse time[1] of the files.

The examination focused on whether files with specific lifetimes are accessed more or less frequently. This analysis aimed to establish a pattern, such as, "if a file is older than a certain number of days, it is likely to be accessed less frequently." For this analysis, only files with both creation and deletion dates were considered, which constituted 44% of all files. This limitation is also why the maximum lifetime depicted in the figures is approximately 80 days.

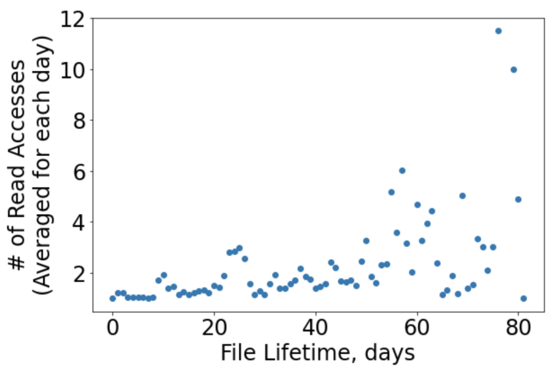
An inherent bias was detected in the data due to the inclusion of accesses from the last month. This introduced a bias: files with a previous reuse distance longer than 60 days were more likely to be accessed. Moreover, if the previous reuse distance is around 80 days, the subsequent reuse distance cannot exceed 10 days. To counter this bias, all accesses occurring within the last month were excluded.

The first result is presented in Figure 4.2a and shows the number of file accesses in relation to their lifetime. It was observed that while files with longer lifetimes tend to be accessed more times, yet this effect becomes less pronounced when analysing the rate (the number of accesses divided by the lifetime), as depicted in Figure 4.2b. Therefore, no clear dependency that could be useful was not present.

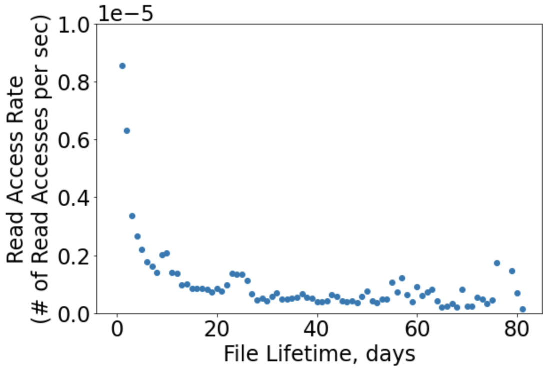
Impact of Previous Access Times on File Reuse

This section examines the potential impact of two features on the target variable: the time distance between the current access and the most recent access to the same file (previous file access time distance), and the time distance between the current access

[1]The reuse time of the files refers to the distance between the current access and the next one to this file.



(a) Number of file accesses depending on the lifetime.



(b) Read access rate depending on the lifetime.

Figure 4.2: Comparative analysis of access frequency and access rate over time.

and the most recent access to any file from the same dataset (previous dataset file access time distance).

The initial objective was to try to predict whether a file would be reused within the subsequent month. In the filtering stage of this task, the last month of trace data was excluded to facilitate the construction of the target variable, resulting in ∼67% of records being retained. Table 4.2 summarises the percentages of records retained for different analyses and the composition of the training datasets.

Analysis Type	**% of Records Retained**	**Labelled as "Accessed"**	**Labelled as "Not Accessed"**
Previous File Access Time Distance	49.72%	23,978,636	3,721,268
Previous Dataset Access Time Distance	63.43%	27,369,345	7,971,000

Table 4.2: The table provides a breakdown of the retention rates and composition of the training datasets used in the study of file reuse predictions within a month based on the previous access times. "Previous File Access Time Distance" measures the time between the current and last access of the same file, and "Previous Dataset Access Time Distance" measures the time between the current access and the last access of any file within the same dataset. For each analysis type, the table shows the percentage of records retained after filtering out the last month's data and constructing the feature, along with the count of files labeled as "Accessed" or "Not Accessed" based on their reuse within the subsequent month.

As shown, the data filtering process significantly reduced the number of access records available for analysis but left enough entries to perform a thorough analysis.

The observations from Figure 4.3 reveal that the two distributions do not significantly differ and follow similar patterns when normalised. Notably, there are pronounced spikes in reaccess frequencies at intervals of less than one second, less than one minute, around one hour, and less than one month. These spikes are also observed in the distribution plots of the general reuse time, suggesting a consistent pattern across different measures of file access and reuse. The pronounced spikes at very short intervals (less than one second) are likely artefacts of the system's operation, rather than reflecting useful access patterns. Although these spikes have minimal impact on caching performance, it is important to acknowledge their source when studying data access patterns. For a more accurate analysis, data should be aggregated based on one logical access, rather than including these artificial "touch" operations, which are likely the source of these short-term spikes.

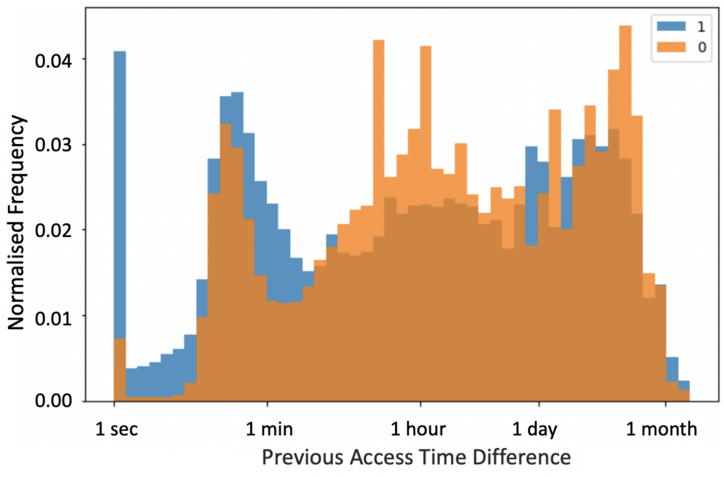


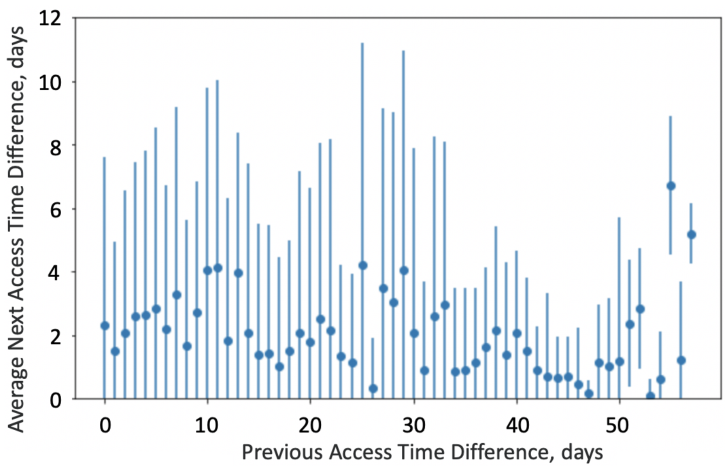
Figure 4.3: Overlay histogram comparing the normalised frequency of previous data access times on a log scale, distinguishing between classes "1" for data accessed and "0" for data not accessed during the target period, across time intervals ranging from one second to one month.

Examining the distribution of the actual reuse distance within the next month, based on the previous access distance (as shown in Figure 4.4a) and the previous access to the same dataset (illustrated in Figure 4.4b), reveals no discernible correlation. In summary, while there are typical values observed, they do not exhibit a correlational relationship.

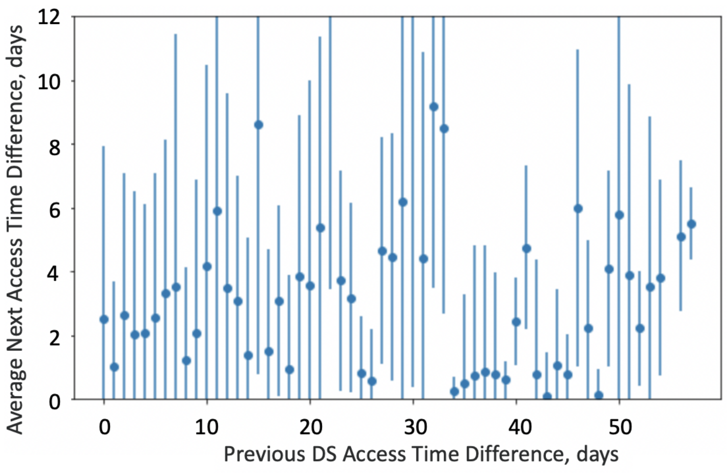
4.4.2 The Choice of the Predictive Model

Machine Learning and Statistical Methods for Predictive Analysis

When deciding on a predictive model for this work, the choice was between machine learning (ML) models and statistical approaches. ML models were selected, primarily



(a) Correlation between 2 consequent accesses to the same file.



(b) Correlation between next file access and previous dataset access.

Figure 4.4: Detailed correlations between the time differences of previous and subsequent file and dataset accesses. Panel (a) illustrates the relationship between two consecutive accesses to the same file, and Panel (b) shows how the time differences between accesses in the same dataset relate. Both plots display the time differences in days across a temporal span of 60 days.

due to the complexity and size of the data. While statistical methods can be applied to large datasets, they often struggle with computational complexity and scalability. ML algorithms, on the other hand, are inherently equipped to handle such datasets efficiently. They benefit from parallel processing libraries and other computational advancements, which make them more suitable for big data.

Furthermore, ML algorithms excel in predicting outcomes and generalising to unseen data. They are adept at automatically identifying patterns and relationships within large datasets. In contrast, statistical approaches tend to focus more on understanding relationships between variables, making inferences about populations, and providing measures of uncertainty. Given the predictive nature of this research problem, this distinction makes ML a more appropriate choice.

Evaluating and Comparing Diverse Regression Models: Preliminary Tests and Insights

This analysis focuses on tree-based learning methods, particularly Decision Trees and their ensembles, Random Forests (RF) and Gradient Boosting Trees (GBT), due to their superior performance in the preliminary tests (Table 4.3). Ensemble learning is a method that combines multiple machine learning models to improve predictive accuracy and reduce the risk of choosing a sub-optimal model. Such an approach tends to make more accurate predictions than any individual model.

There are various types of ensemble learning methods [78], the main ones are:

- **Parallel Ensemble Learning (Bootstrap Aggregating, or Bagging)**: Bagging involves training multiple models in parallel, each on a random subset of the training data. The final prediction is typically an average (for regression) or a majority vote (for classification). Random Forest is a classic example of bagging, where many decision trees are trained on different data subsets and their predictions are averaged.

- **Sequential Ensemble Learning (Boosting)**: In boosting, subsequent models are trained by focusing more on the instances that previous models misclassified or gave higher errors. The idea is to improve the model iteratively by focusing on the harder cases. Boosting algorithms include Gradient Boosting Trees, AdaBoost and XGBoost.

- **Stacking**: Stacking involves training multiple models (usually of differing types) on the same dataset and then using another model, often called a meta-learner, to synthesize the outputs of the individual models. The base models are trained on the complete dataset, and their predictions form the input features for the meta-learner, which aims to learn how best to combine these predictions to make a final prediction.

One should mention, that aggregated or ensemble models do not inherently outperform their single-model counterparts in every scenario. Usually, their advantage is most notable in reducing variance and improving reliability in models prone to instability.

Various machine learning regression techniques from the `pyspark.ml.regression` library were assessed [79], such as Linear Regression, Generalised Linear Regression, Decision Tree Regressor, Random Forest Regressor, Gradient-Boosted Tree Regression, Isotonic Regression, and Factorization Machines Regression. A basic dataset was created with three features—file size, last opening duration[2], and the difference in last opening time—to predict the logarithm of the next reuse distance. This formulation of the predicting target was chosen over predicting the reuse time value to improve interpretability and differentiate between files reused within varying time frames. Although the computational effort is nearly the same, these two values are easily interchangeable.

These three features were normalised for the regression task and used *Root Mean Square Error* (*RMSE*) as the evaluation metric, as shown in Table 4.3. Additionally, based on the distribution of the target variable, which aligns with the distribution of previous access time (illustrated in Figure 4.3), all data where the target variable was less than 10 seconds were excluded. This step, detailed in the "*RMSE* (After Noise Reduction)" column of Table 4.3, enhanced the results. However, this improvement could simply be attributed to the reduction in the output range.

Method	***RMSE* (Initial)**	***RMSE* (After Noise Reduction)**
Linear Regression	4.36	3.73
Generalised Linear Regression	4.35	3.71
Decision Tree Regressor	3.32	3.01
Random Forest Regressor	3.51	3.11
Gradient-Boosted Tree Regression	3.22	2.93
Isotonic Regression	4.92	3.71
Factorization Machines Regression	8.73	9.15

Table 4.3: Comparison of *RMSE* scores for different regression methods before and after noise reduction. This table evaluates several ML regression techniques from the `pyspark.ml.regression` library. The analysis is based on three normalised features: file size, last opening duration, and the difference in last opening time. The goal is to predict the logarithm of the next reuse distance of files.

[2]The last opening duration refers to the time interval between the closing and opening time of the last access operation to a file.

Our findings demonstrated that tree-based methods (Decision Tree, Random Forest, and Gradient-Boosted Tree) were superior in performance.

Gradient Boosted Trees Gradient Boosted Trees employ an iterative approach to train a series of decision trees. In each iteration, it uses the combined predictions of the existing trees to assess each training entry, comparing these predictions with the actual labels. Entries that are poorly predicted are then given more weight in the subsequent iteration, guiding the next tree to focus on these errors for correction.

One key characteristic of GBT is its ability to achieve superior results with a relatively smaller number of trees compared to other methods. However, this comes at the cost of increased time for model construction. It is equally important to note that GBT, unlike bagging ensemble methods such as Random Forest (RF), is more susceptible to overfitting.

Given the large dataset and the need for a method that is both rapid and robust against overfitting, GBT (Gradient Boosted Trees) was omitted from the analysis. This decision was based on prioritising methodologies that align better with the data characteristics and analysis goals.

Decision Trees and Random Forest

After careful study of existing ML techniques, the focus was placed on Decision Trees and their ensembles, Random Forests. Decision trees and random forests are both powerful machine-learning algorithms that can be used for classification and regression tasks.

A decision tree is a tree-like model that makes decisions based on a series of binary splits on the input features (Figure 4.5). Each split divides the data into subsets based on the value of a particular feature until the final leaves of the tree represent the predicted output for each subset. Random forests are an ensemble of decision trees that are trained on subsets of the data and subsets of the features and then aggregated to produce a final prediction. Each tree in the forest is grown independently, and the final prediction is determined by a majority vote or an average of the individual predictions.

Decision trees are easy to interpret and can handle non-linear relationships between features and the target variable. However, they are prone to overfitting and may not generalize well to new data. Random forests address the overfitting issue by aggregating the predictions of multiple trees. They are also able to handle high-dimensional data and noisy data. However, they may be more complex to interpret than a single decision tree.

For the task of predicting future file accesses based on previous data, random forests were chosen because they can handle high-dimensional data and are robust to noise. This

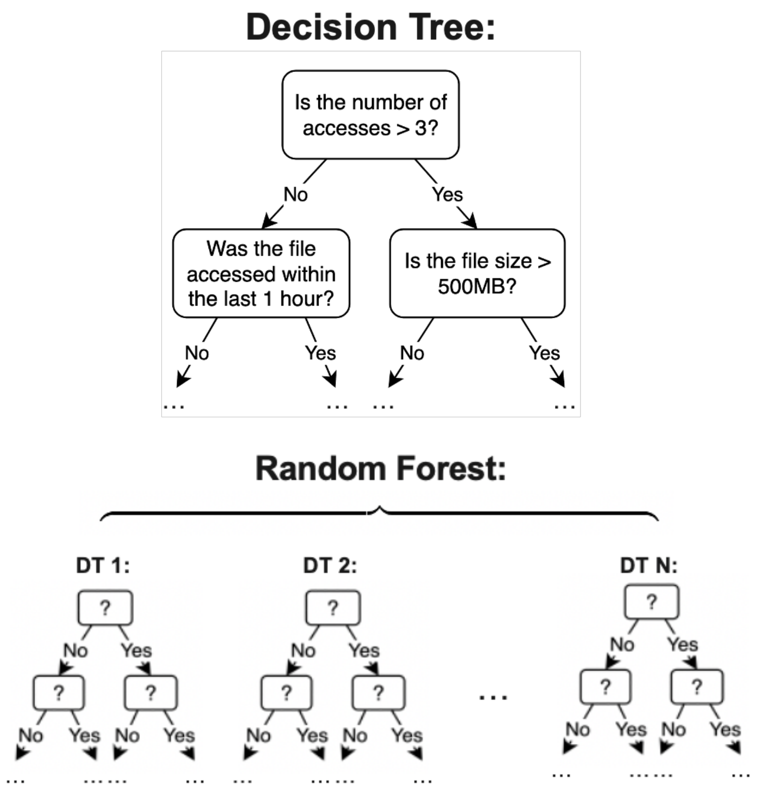

Decision Tree:
Is the number of accesses > 3?
No
Yes
Was the file accessed within the last 1 hour?
Is the file size > 500MB?
No
Yes
No
Yes
Random Forest:
DT 1:
DT 2:
DT N:
?

Figure 4.5: Decision Tree and Random Forest schematic representation.

method was selected due to its effectiveness in handling both classification and regression tasks and its capability for parallelization. Additionally, exploring the importance of different features in the prediction task is readily facilitated by random forests, making it a valuable tool for feature analysis.

4.4.3 Training Decision Trees and Random Forests

Preparing the Data

Decision Trees and Random Forests typically do not require normalisation of data due to their operational nature. These algorithms make decisions by splitting on feature values, focusing on the ordering of the data rather than their scale. Therefore, standardising or normalising features, which alters their scale but not their ordering, does not significantly impact the performance of these models.

Balancing is essential for the target variable (label) to prevent the model from being biased towards the more represented classes (more dense areas of the popularity distribution). Although balancing was not implemented for the regression task, mainly due to the complexity involved in such contexts, it was also considered that the inherent bias towards more frequent reuse distances might be advantageous in regression scenarios. However, for classification tasks, balancing was diligently applied to ensure a more equitable and accurate representation across various classes.

Furthermore, as described in the preliminary tests with different regression models, the decision was made to predict the logarithm of the reuse time, rather than the raw value, to improve the model's performance and interpretability.

Random Forest Implementation

Suppose a Random Forest employs K Decision Trees. Each tree is built from a random subsample of the original dataset (D), ensuring diversity in the analysis. During the construction of each tree, a subset of features (F) is randomly chosen for node splitting. F was set to equal the square root of the total feature count, which is typical in regression tasks. The deciding feature will be determined based on the amount of entropy it creates.

The node impurity is a measure of the homogeneity of the labels at the node. It is calculated as follows:

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \mu)^2 \tag{4.1}$$

where y_i is a label for an instance, N is the number of instances and μ is the mean given by

$$\frac{1}{N}\sum_{i=1}^{N} y_i \tag{4.2}$$

The impurity decrease (also called "information gain") is the difference between the parent node impurity and the weighted sum of the two child node impurities. The equation is the following:

$$\frac{N_t}{N}\left(\text{impurity} - \frac{N_{t_R}}{N_t} \times \text{right\_impurity} - \frac{N_{t_L}}{N_t} \times \text{left\_impurity}\right) \tag{4.3}$$

where N is the total number of samples, N_t is the number of samples at the current node, N_{t_L} is the number of samples in the left child, and N_{t_R} is the number of samples in the right child.

There are several issues with the procedure of recursively partitioning data to construct decision trees (DTs). Allowing the trees to grow until each leaf contains only a single observation leads to significant problems. First, this process is time-consuming, as the number of splits that need to be tested grows exponentially with the increase in the number of leaves in the trees. Second, there is a critical point where continuing to split nodes into smaller child nodes should be halted; overly complex trees with many branches and leaves tend to overfit the training data.

In addressing these challenges, various stopping criteria are defined to govern the growth of decision trees in both Decision Tree and Random Forest algorithms. These criteria, integral to controlling overfitting and improving the model's generalization capabilities, are readily adjustable in PySpark, allowing for tailored model complexity and enhanced predictive performance.

To conduct the experiments, the libraries used are `pyspark.ml.classification.RandomForestClassifier` for classification tasks and `pyspark.ml.regression.RandomForestRegressor` for regression analysis. The following are the key stopping criteria, along with their corresponding parameters in the PySpark implementation, if any:

- **Maximum Number of Leaves**: The growth of the tree is halted once it reaches a predefined number of leaves.
- **Maximum Depth**: The tree expansion is limited to a specified depth, defined by the `maxDepth` parameter in PySpark.
- **Minimum Samples for Split**: This criterion prevents further splitting of a node if it contains fewer observations than a specified threshold, ensuring that each leaf has

at most `N` entries.

- **Minimum Samples per Leaf**: A split is not performed if it results in child nodes having fewer observations than a predefined limit. This is determined by the `minInstancesPerNode` parameter in PySpark.
- **Minimum Impurity Decrease**: Splits are only executed if they lead to a significant decrease in impurity, compared to the parent node's impurity. This is quantified by the `minInfoGain` parameter in PySpark, whereby a node will be split if this split induces an impurity decrease greater than or equal to this value.
- **Purity of Nodes**: The process is naturally terminated once all observations in a node have either the same target value, indicating a pure node, or identical feature values.

A decision tree's expansion ceases when at least one of these criteria is satisfied, thus preventing the model from overfitting and ensuring better generalization. The number of trees in a forest is defined by the `numTrees` parameter.

Split candidates of Continuous features In scenarios involving small datasets, the process of identifying split candidates for each continuous feature typically involves using the feature's unique values. This approach is straightforward: the feature values are sorted, allowing the algorithm to employ these sorted, unique values as split candidates. This sorting facilitates faster tree calculations.

However, for large datasets distributed across multiple machines, sorting feature values becomes computationally expensive and less feasible. To address this, the pySpark implementation adopts a different strategy. It approximates the set of split candidates by performing a quantile calculation on a sampled subset of the data. This approach effectively generates ordered splits, which are then grouped into "bins". The granularity of these bins can be controlled using the `maxBins` parameter, allowing to specify the maximum number of bins to be created. This method provides a balance between computational efficiency and the precision of the split candidates in large, distributed datasets.

Addressing Overfitting in Random Forest Models It is important to highlight the two primary sources of randomness in the construction of a Random Forest, both aimed at decreasing the variance of the forest estimator. Firstly, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.

This process is controlled by parameters `subsamplingRate` and a boolean `bootstrap`, which can be adjusted as needed. The randomness injected into forests results in decision trees with decoupled prediction errors. By averaging these predictions, some errors can effectively cancel out, enhancing the overall accuracy of the model.

Furthermore, as previously mentioned, when splitting each node during the construction of a tree, the best split is determined from a random subset of all input features, controlled by the `featureSubsetStrategy` parameter. This random selection of features contributes further to reducing the variance of the model.

During the studies, it has been observed that the Random Forest model with default configuration exhibited marginally lower accuracy compared to a single Decision Tree (Table 4.3). This discrepancy is attributed to the default setting of the RF model, where the number of features considered for each split is the square root of the total number of features. However, upon modifying this parameter to `featureSubsetStrategy = ■all■`, which considers all features for each split, the performance of the RF model significantly improved, surpassing that of the single DT. This adjustment highlights the impact of the feature selection strategy on the efficacy of the RF model.

It is also worth noting that Random Forest's inherent randomness in feature selection across trees offers a form of built-in cross-validation, reducing the necessity for external cross-validation methods.

Fine-Tuning Hyperparameters of Random Forest Models

The optimisation of hyper-parameters in Random Forest models was also examined. Here, some of the key findings are presented.

The primary parameter for adjustment in Random Forests is the number of trees in the forest. While a larger number generally improves model performance, it also increases computational time. Importantly, the improvement in results plateaus beyond a certain number of trees. As expected, in scenarios where `numTrees`, the Random Forest model essentially replicates the results of a Decision Tree.

Another influential feature is the size of the random feature subsets for node splitting. Lower numbers of considered features can reduce variance but increase bias. Importantly, the performance of a model can be adversely affected by the inclusion of irrelevant features, particularly when these features are randomly selected for splits (when `featureSubsetStrategy` differs from "`all`").

Fully grown trees, with unlimited tree depth and combined with fully split nodes (`maxDepth=None` combined with `minInstancesPerNode=1` can lead to good results, although they may not be optimal and can result in high memory usage. These values should be cross-validated for best performance.

The study found that using the full dataset can lead to repetitive trees, diminishing diversity and performance. However, too small a sample size can increase errors. The `bootstrap` parameter, a method for sampling data points, significantly affects performance.

Eventually, the most effective combination of parameters identified for this testing task was: Eventually, the most effective combination of parameters identified for this testing task was as follows: `featureSubsetStrategy = "all"`, `subsamplingRate = 0.01`, `maxBins = 256`, `maxDepth = 15`, `numTrees = 20` (default), and `bootstrap = True` (default). This configuration yielded a *RMSE* on test data of 2.88 (for comparison, refer to Table 4.3), and a *RMSE* on training data of 2.88.

4.4.4 Reformulating the Problem: Watermarks Training

After encountering a performance plateau in fine-tuning the regression model, it seemed advisable to transition to a different predictive approach. Instead of predicting the reuse distance between consecutive accesses, a threshold was established (a fixed moment in time) and predicted the time from this threshold to the next access, using previous access data and other available parameters. As detailed in this subsection, this redefined prediction task yielded significantly improved results. Furthermore, it facilitated more realistic integration into an operational cache system. Rather than predicting for each file upon every access, predictions were only necessary for all files in the cache when the high watermark was reached (refer to Subsection 4.5.2 for additional details).

To develop this new approach, it was necessary to construct an entirely different training dataset. The dataset was partitioned into features and target sets based on a time threshold. This divided the 3-month trace into approximately 75 and 15 days, corresponding to 79.33% and 20.67% of all accesses, respectively.

Reformulating the task in this way allowed to greatly enhance the list of features used for the training. In total, 18 features have been tailored, including file and dataset sizes, frequencies, recency of file accesses, access durations, and dataset-related characteristics. The full list:

- file size (physical storage size);
- recency[3] of the most recent file access;
- recency of the second-to-most-recent file access;
- recency of the file creation;

[3]Time interval between `NOW` and the corresponding access.

- recency of the first file access;
- number of file accesses within the last day;
- number of file accesses within the last week;
- number of file accesses within the last month;
- total number of file accesses since file creation;
- duration[4] of the most recent access;
- time difference between file creation and most recent access;
- time difference between file creation and first access;
- time difference between first and most recent file accesses;
- difference between the average access duration and the duration of the most recent access;
- difference between the maximum access duration and the duration of the most recent access;
- dataset size (physical storage size);
- dataset size (number of files); and
- recency of the most recent dataset access.

As a start, the prediction has been run with all the features possible (including the ones that are not available for the real trace, and the dataset-related ones). This approach was adopted to assess the potential for obtaining meaningful results and to determine whether the inclusion of non-existent features would be justified.

Working with this dataset, two prediction tasks have been formulated:

- **Regression Task:** Predict the reuse distance of the file from the time stamp to the closest access. (As in the previous subsection, the log of the distance is predicted for the same reasons).
- **Classification Task:** Calculate the probability of whether a file will be accessed in the next 15 days (binary classification task).

The same RF model architecture and feature set have been used for both classification and regression tasks, although different prediction goals implied slightly different filtering steps. The filtering was mainly due to the use of features not present for all files.

[4]time interval between file opening and closing of the corresponding access.

For the regression task, the filtering process consisted of choosing files that have at least 2 accesses, and files with accessible creation dates. Additionally, each file needed to have at least 1 access within the target dataset. Starting with over 10 million files, this filtering process resulted in 342,878 files (only 2.89% of the original number) containing all the necessary features for training the ML model (refer to Table 4.4 for task type "Both" and "Regression").

Filtering Stage	**Task Type**	**Number of Files**	**Percentage of Files**
Before the threshold	Both	10,556,833	88.98%
With 2+ accesses	Both	4,719,609	39.78%
With creation time	Both	2,378,787	20.05%
With target access	Regression	342,878	2.89%
After target balancing	Classification	686,942	5.79%

Table 4.4: Filtering reduction rates of the training dataset for regression and classification tasks, which aim to predict file access within the next 15 days using Random Forest model. This table illustrates the sequential reduction of the dataset from over 10 million files to subsets suitable for specific ML tasks. The filtering criteria, based on target and feature requirements, include having more than two accesses and a known creation time. For regression, at least one relevant access within the dataset reduced the count to 342,878 files (2.89% of the original). For classification, after balancing the classes "Accessed" and "Not Accessed," 686,942 files (5.79% of the original) remained.

For the classification, instead of excluding the files that were not used in the next 15 days, they have been used to train the model, but the balancing of the prediction classes (refer to Subsection 4.4.3) needs to be performed. The final dataset comprised 686,942 files, accounting only for 5.79% of the original number (See also task types "Both" and "Classification" in Table 4.4).

Note. After the balancing of the target for the classification task, the *RMSE* looks much worse, however, the "*AreaUnder*" results have improved, compared to the unbalanced dataset. This was expected. When the training (and test) datasets are unbalanced, the model is biased towards more frequent results, which improves the accuracy, the *RMSE*, but it actually learns wrong patterns. so the area increases. Balancing actually improved the model even though the accuracy dropped.

Evaluation metrics The *RMSE* has been employed as the evaluation metric for regression, and *RMSE*, *Area Under the ROC* (*Receiver Operating Characteristic*) curve (*AreaUnderROC*), and *Area Under the PR* (*Precision-Recall*) curve (*AreaUnderPR*) for classification. The use of *RMSE* as a regression-specific metric was justified by the caching model's utilisation of the exact predicted probability. Conversely, *AreaUnderROC* and *AreaUnderPR* are standard classification-specific metrics.

The *ROC* curve itself is a plot of the *True Positive Rate* (*TPR* or *Sensitivity*) against the *False Positive Rate* (*FPR* or $1 - Specificity$) at various threshold settings for a classification model. *AreaUnderROC* quantifies the trade-off between the *TPR* and *FPR*. The *PR* curve, on the other hand, plots precision (the proportion of true positive predictions among all positive predictions) against recall (the proportion of true positives among all actual positives), focusing on the performance of a classifier on the positive class, which is particularly useful in imbalanced datasets.

The Units

Although the prediction involves the logarithm (*log*) of a time value that is measured in seconds, the result of the *RMSE* will be unitless. This is because the logarithm itself does not have units; it is a pure number that represents the logarithmic relationship between the original time value and the chosen base.

The *AreaUnderROC* and *AreaUnderPR* are also unitless metrics. They measure the performance of a binary classification model, and the value ranges from 0 to 1, where 0 indicates poor performance (the model makes all predictions incorrectly), and 1 indicates perfect performance (the model makes all predictions correctly).

4.4.5 Results of the Prediction Models

Through model tweaking and hyperparameter tuning, some encouraging results were achieved, summarised in Table 4.5 and Table 4.6 (for both, column “M1”).

The fact that the error on training and testing is similar is what one would expect, this means that there is no overfitting. Additionally, it is observed that Random Forest significantly outperforms Decision Trees, with performance metrics almost approaching the ideal value of 1.

One can note that these results look significantly better than the previous formulation of the problem (refer to Table 4.3). These compelling results validate the effectiveness of the ML approach and open the way for enhancing cache eviction policies.

To get a deeper look into the model’s prediction capacities, several key metrics were plotted. In regression analysis, the focus was on examining whether the distribution of the target variable changes (as seen in the actual distributions in Figures 4.6a and 4.7a versus the predicted ones in Figures 4.6b and 4.7b). The observation that the main patterns (peaks and troughs) are retained confirms the quality of the prediction.

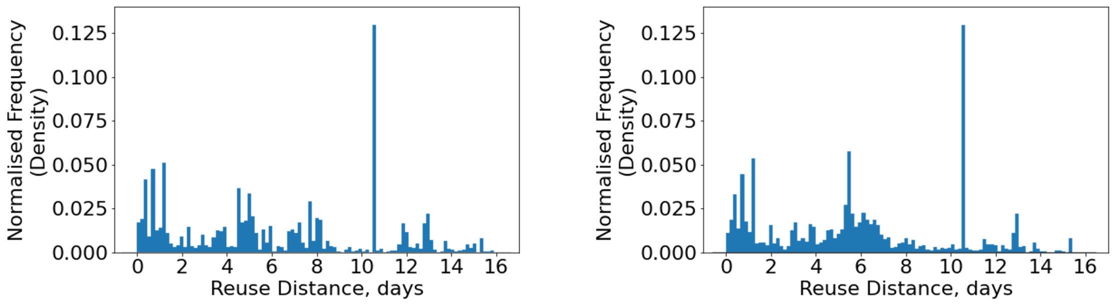
Figure 4.8 indicates that the model tends to be more accurate the bigger the target label is.

	M1	**M2**	**M3**
Regression			
RMSE on the training data (70%)	0.291	0.387	0.432
RMSE on the test data (30%)	0.346	0.431	0.460
Classification			
RMSE on the training data (70%)	0.108	0.128	0.196
RMSE on the test data (30%)	0.119	0.142	0.196
AreaUnderROC on the training data	0.998	0.996	0.988
AreaUnderROC on the test data	0.998	0.994	0.988
AreaUnderPR on the training data	0.998	0.995	0.987
AreaUnderPR on the test data	0.997	0.993	0.987

Table 4.5: ML model performance comparison (Random Forest). **M1** - The initial full model with 18 features. **M2** - the model excludes dataset-related features, so 15 features are left. **M3** - the reduced model with only really existing in the trace features (6 features).

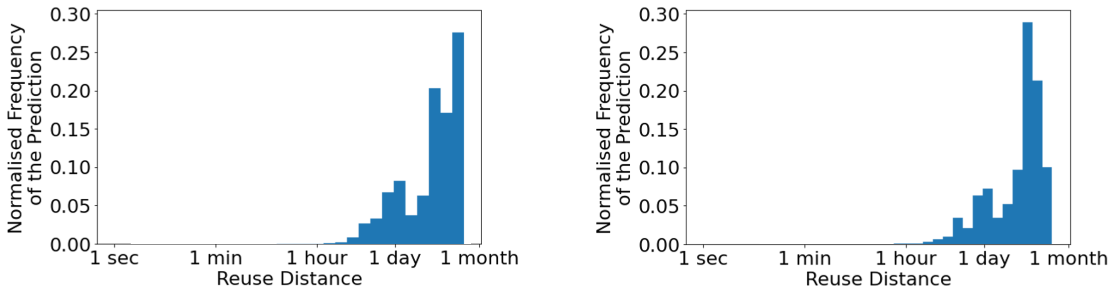
	M1	**M2**	**M3**
Regression			
RMSE on the training data (70%)	0.323	0.452	0.500
RMSE on the test data (30%)	0.398	0.509	0.523
Classification			
RMSE on the training data (70%)	0.120	0.142	0.210
RMSE on the test data (30%)	0.136	0.160	0.216
AreaUnderROC on the training data	0.974	0.952	0.938
AreaUnderROC on the test data	0.972	0.949	0.937
AreaUnderPR on the training data	0.975	0.954	0.921
AreaUnderPR on the test data	0.974	0.952	0.920

Table 4.6: ML model performance comparison (Decision Tree). **M1** - The initial full model with 18 features. **M2** - the model excludes dataset-related features, so 15 features are left. **M3** - the reduced model with only really existing in the trace features (6 features).



(a) Actual. (b) Predicted.

Figure 4.6: Regression analysis: comparison of Actual vs. Predicted reuse distance distributions. This figure presents the normalised frequency distributions for the actual and predicted reuse distances of files over a period of 16 days. Panel (a) shows the actual observed reuse distances, and Panel (b) depicts the predicted reuse distances derived from the regression model. Each distribution is plotted on a linear scale to provide a clear view of the frequency and variance in file reuse patterns, illustrating the model's ability to mimic real-world usage behaviour.



(a) Actual. (b) Predicted.

Figure 4.7: Regression analysis: Actual vs. Predicted distribution of reuse distances. This figure contrasts the actual and predicted distributions of reuse distances for files, ranging from one second to one month. Panel (a) displays the actual observed frequencies of reuse distances, while Panel (b) shows the predicted frequencies based on the regression model. Both distributions are normalised and plotted on a logarithmic scale to highlight differences across a broad range of time intervals.

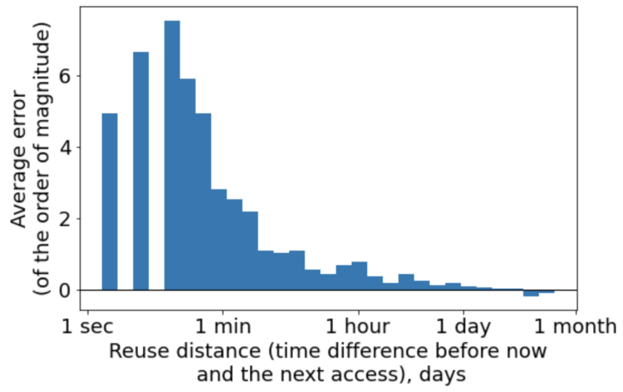


Figure 4.8: Regression model error distribution by reuse distance. This figure details the average error magnitude across different predicted reuse distances, from one second to one month. The histogram shows that errors tend to decrease as the reuse distance increases, indicating varying levels of prediction accuracy across short and long reuse intervals.

For classification, the distribution of the target classes in the training dataset is levelled out (due to the balancing). The distribution in the prediction is shown in Figure 4.9.

Figure 4.10 indicates that the model tends to have false positives rather than false negatives. It is possible to take a raw prediction (the probability that an entry belongs to this class) and set a custom threshold to determine the final prediction. In the case, the standard threshold parameter (0.5) gave a sufficiently good result. By tuning this parameter of the classifier, one can achieve a better ratio between false positives and false negatives.

4.4.6 Feature Importance

To gain insights into the Random Forest models' workings, the feature importance was examined, a valuable attribute that RF provides [80]. There are several methods to measure it, including permutation feature importance and accuracy-based importance, which relies on out-of-bag samples from each decision tree in a forest. In this study, Gini-based importance was selected for its efficiency. One advantage of the Gini-based importance is that the Gini calculations are already performed during training, so minimal extra computation is required. The feature importance values are available as fields of the trained regression and classification models.

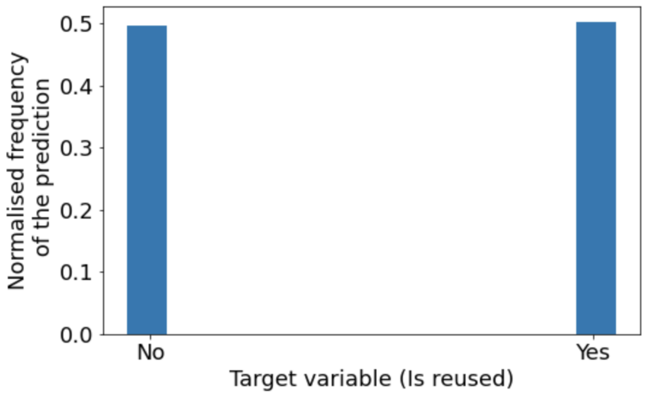


Figure 4.9: Classification model: distribution of predicted classes. This figure presents the normalised frequency of predictions for the target variable “Is reused” in a classification model, showing a comparable likelihood of files being predicted as “Accessed” and “Not Accessed,” correspondingly. The graph highlights the balanced distribution of predicted outcomes in the model, which corresponds to the balanced distribution in the training dataset.

Gini-Based Importance

When a tree is built, the decision about which variable to use to split at each node uses a calculation of the Gini impurity. For each variable, the sum of the Gini decrease across every tree of the forest is accumulated every time that variable is chosen to split a node. The sum is divided by the number of trees in the forest to give an average. The scale is irrelevant: only the relative values matter.

Figure 4.11 demonstrates the results of gini-based feature importance for the Random Forest regression model. Several interesting observations could be made. The importance of dataset-specific metrics is apparent. They tend to contribute the most to the model’s decisions. The frequencies of the previous access do not seem to be too relevant. Additionally, the duration of the last access in comparison to the duration of the previous access to the same file tends to contribute the least.

It is important to note, that each time the model is retrained, the actual values of the feature importance slightly change, but their relative position and contribution stay more or less the same.

Figure 4.12 demonstrates the importance of the same features, but in the binary classification case of Random Forest. Firstly, it is clear that the distribution is very different. Very apparently, the recency of the last access contributes hugely to the result, it is at least 5 times more important than any other feature.

Overall, dataset-related features hold significant importance in both models. This

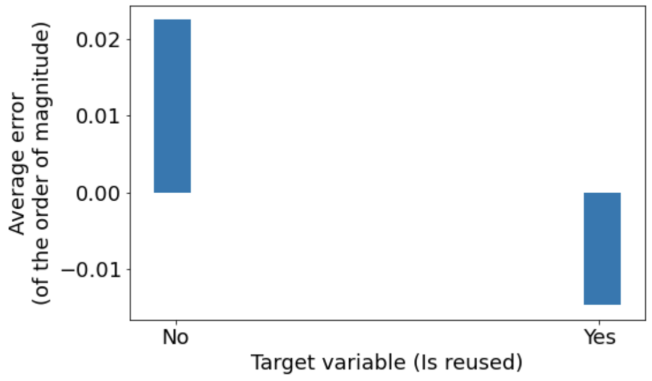


Figure 4.10: Classification model: error magnitude depending on the target class. This plot displays the average error magnitude for each class ("Yes" corresponding to "Accessed" and "No" - "Not Accessed") within the classification model. Classification model error depending on the target class

suggests that the dataset a file belongs to substantially influences its likelihood of being reused. Additionally, recency-based features, such as the last file and dataset access times, play a critical role in models. They indicate a file's relevance and potential future reuse.

Feature Importance by Exclusion

Another way to understand the importance of a specific feature is by completely excluding it from the training set. This is significantly more computationally expensive, but gives a clearer indication on whether adding this feature improves the overall prediction accuracy or not.

Six most influential features for the regression prediction task have been taken into consideration:

- recency of the most recent file access;
- time difference between file creation and most recent access;
- number of file accesses within the last month;
- file size (physical storage size);
- recency of the file creation; and
- recency of the first file access.

Combined, these features contribute almost 70% to the model prediction decision.

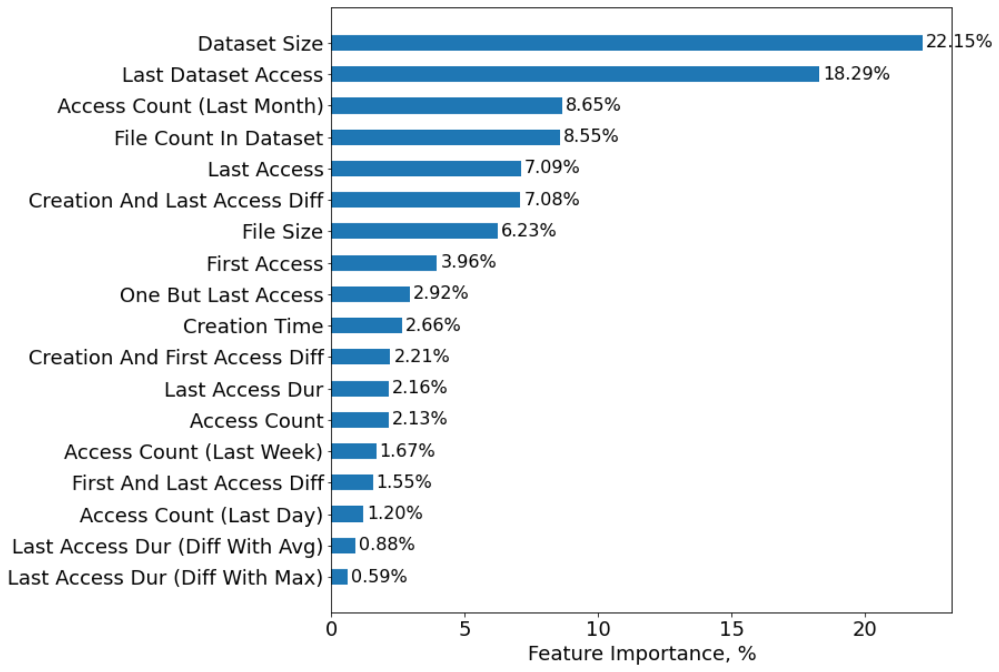


Figure 4.11: Feature importance for the RF regression model. Features are ranked by their contribution to the final predictions.

The results are presented in Table 4.7 ("M4") in comparison to the initial full model ("M1"). As expected, for the RF model the result got worse, but remained very acceptable. This indicates that if the number of features needs to be reduced, the between accuracy and lightweight of the model will be rather good.

	M1	**M4**
Random Forest		
RMSE on the training data (70%)	0.291	0.387
RMSE on the test data (30%)	0.346	0.431

Table 4.7: Regression model performance comparison. **M1** - The initial full model with 18 features. **M4** - the model with the most important features only (6 features)

A similar analysis has been performed for dataset-related features, for regression and classification, as well as Random Forest and Decision Tree models. The three dataset-related features (dataset size, number of files in the dataset, and the time difference with the last access to this dataset) have been excluded and the model has been retrained and the the model performance has been reevaluated, column "M2" in Tables 4.5 and 4.6 show the results. Although this exclusion did not fundamentally undermine the models' performance, some deterioration of the results can be observed.

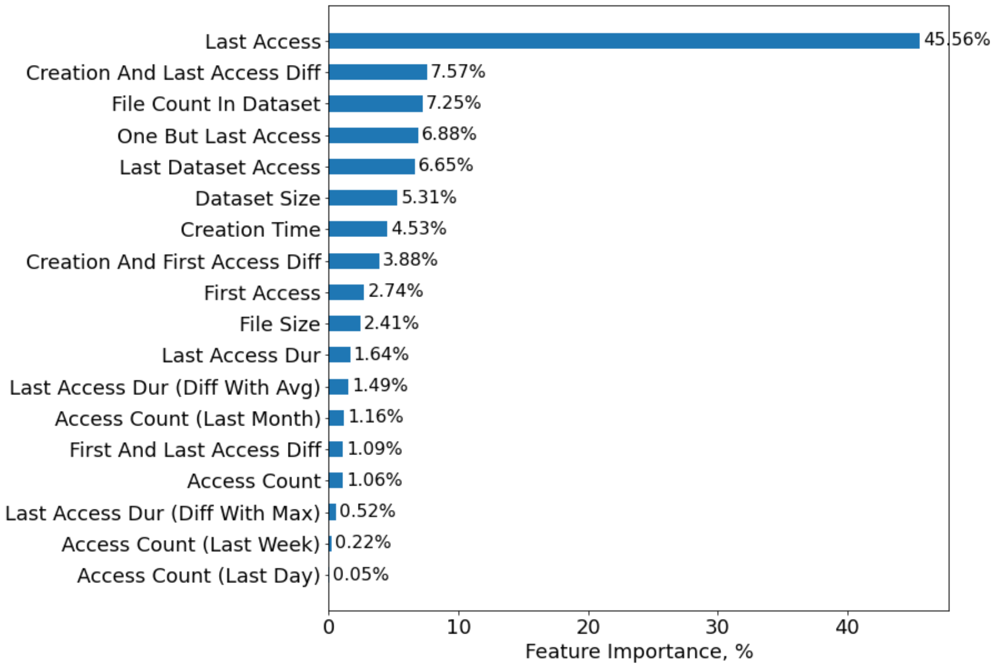


Figure 4.12: Feature importance for the RF classification model. Features are ranked by their contribution to the final predictions.

4.5 Integration into the Caching Policies

4.5.1 Condensing Model

Some of the features that have been used in the previous section will realistically not be available. Although in theory, it is possible to keep this kind of information in the metadata and transferring a bit of metadata with the file when sent to the cache is not too expensive, for example, Rucio maintains more information about the files than the storage systems. In the real case, the situation would be comparable, the first encounter of a cache with a given file would provide the cache with the very first information about the characteristics of this file.

Therefore, to extend the applicability of the developed ML models to the entire access trace currently available, they were retrained to operate with a reduced set of features:

- file size (physical storage size);
- recency of the most recent file access;
- duration of the most recent access;
- dataset size (physical storage size);

- dataset size (number of files); and
- recency of the most recent dataset access.

Despite the reduced feature set, the adapted models still demonstrated robust performance in both regression and classification tasks as can be seen in Tables 4.5 and 4.6, "M3". The minimal decrease in performance indicates the resilience and flexibility of the ML models, allowing them to effectively handle the limited number of available features.

4.5.2 Watermarks

The process of integrating predictive capabilities into the cache eviction algorithm involved a series of strategic adjustments to the existing implementation. Instead of carrying out cache deletions on a per-file basis, a more practical approach known as the "Watermarks" method has been adopted. This approach allows the cache's size to fluctuate within predetermined thresholds: a high watermark and a low watermark, which can be configured to suit specific needs. In this particular case, the low watermark is set at 80% of the cache's capacity and the high watermark at 95%.

The purpose of the high watermark is to ensure that there's still sufficient room within the cache to accommodate incoming files while the evaluation process, which determines which files to evict, is in progress. Without this buffer, pending requests would disrupt the system. Meanwhile, the low watermark serves as the limit for the cache purging process. The choice of high and low watermark levels is influenced by the rates at which incoming requests arrive (to maintain adequate space during evaluations) and the time it takes to complete the evaluation.

It's worth noting that setting these watermarks too conservatively can limit the practical cache size, so a careful balance is required and in a production systems needs to be monitored carefully.

To incorporate predictive capabilities into the cache eviction algorithm, when the high watermark is reached, it triggers the predictive process for all files currently residing in the cache. Subsequently, the files are sorted based on their predicted reuse distance or probability of being reused, depending on whether it's a regression or classification scenario. In the case of regression, the actual predicted value is employed for sorting, while in the classification case, the probability of a file being reused within the next two weeks is used. In both cases, the Random Forest-based models are used, as they showed better results.

This approach, although slightly less performing than the original file-by-file method, offers significant advantages. It reduces the frequency of cache cleanup (refer to Table 4.8), leading to decreased CPU load. In fact, as demonstrated in Figure 4.13, this realistic implementation shows minimal performance reduction compared to the original approach, underscoring the effectiveness of the watermarks-based cache management in maintaining cache efficiency while reducing CPU overhead.

Furthermore, the watermarks model conveniently facilitates the integration of predictive models. The prediction process for machine learning models is only triggered occasionally, specifically when the high watermark is reached. This stands in contrast to the constant recalculation of file priorities in the eviction queue by the cache, resulting in an overall enhancement of the caching system's performance.

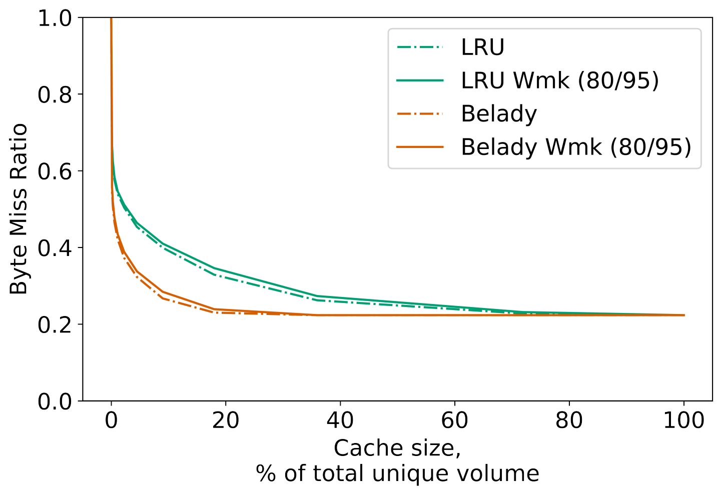


Figure 4.13: Visual comparison of the basic *LRU* and *Belady* cache eviction policies with their watermark-based counterparts (Wmk).

Cache Size	**Number of Cache Cleanups**	**Simulation Time (min)**	**Average *RMSE***
100%	0	13	-
72%	4	16	1.61
36%	18	20	1.79
18%	53	25	1.94
9%	136	36	1.99
4%	319	57	2.12
2%	724	101	2.39

Table 4.8: Impact of the cache size on the number of cache cleanups, simulation time and the average *RMSE* of the Random Forest classification task. As cache size decreases, the number of required cache cleanups increases, extending the simulation time and affecting prediction accuracy (*RMSE*).

Interestingly, *Belady* and *Belady-Watermarks* behave almost perfectly with large cache sizes (seen on Figure 4.13). This is mainly due to the fact that when the cache size is big, much fewer files need to be removed. The first candidates to be evicted are the files that will not be used in this trace again, and whose eviction does not influence the final hit ratios.

4.5.3 Experimental Results and Discussion

The results depicted in Figure 4.14 reveal that the classification model performs somewhat better than the regression model, but still falls short of outperforming *LRU*. One possible explanation for the classification model's superior performance could be attributed to the fact that with small cache sizes (2-5%), the distinction between files likely to be reused within several days versus several weeks or more becomes more crucial. In this context, the classification approach is better equipped to handle such distinctions.

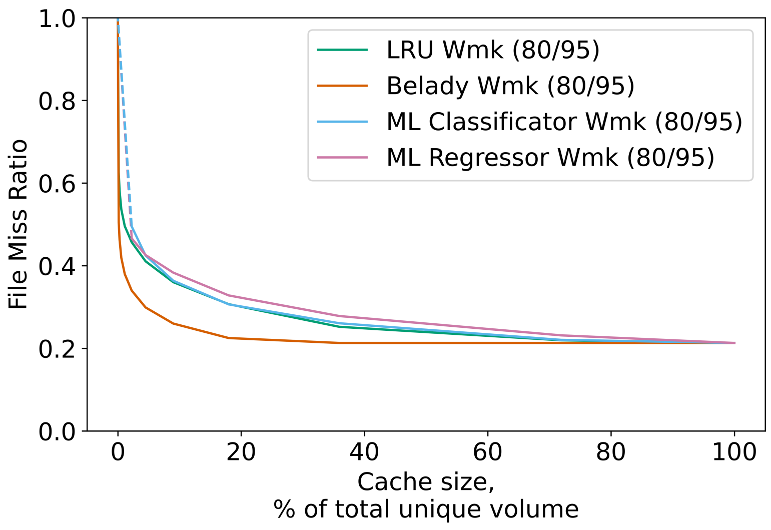
Further analysis has been conducted to understand why predictors with favourable scores did not lead to significant improvements in the cache eviction policy. It became evident that the performance of the trained models on the actual full trace was considerably worse than on the test data. Specifically, the average *RMSE* of the regression model for each cache size was notably higher than the *RMSE* of 0.46 obtained on the test data (as demonstrated in Figure 4.15 and Table 4.8).

Several factors contribute to this discrepancy. Firstly, the model's training exclusively relied on files created within the restricted time frame. Moreover, a significant portion of files within this short time window were used only once (approximately 60%), a scenario that a regression model alone cannot adequately capture. Additionally, the training data underwent specific feature/label cuts, diminishing the diversity of training entries, and the prediction was constrained to a 15-day timeframe.

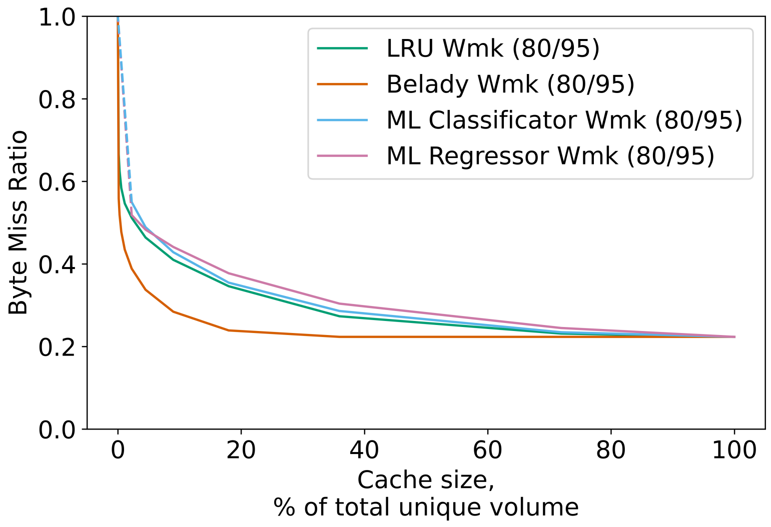
These findings reveal the limitations of the current implementation, while highlighting several directions for further enhancement of the ML-based solutions, such as expanding the training dataset and optimising model hyperparameters. This would allow the models to learn from more diverse and extensive data, which overall can lead to better predictive capabilities. Additionally, by exploring different ML model combinations, potentially more effective ways to leverage the predictive power of ML for cache management could be found.

4.6 Implementation Details and Computational Complexity

For understanding the the computational complexity it is instructive to look at running the simulation for one cache size, which is tightly related to the implementation details



(a) *FMR.*



(b) *BMR.*

Figure 4.14: Visual comparison of *LRU* and *Belady-Watermarks* models, and models integrating regression and classification predictions, along with the theoretical optimum lower bound.

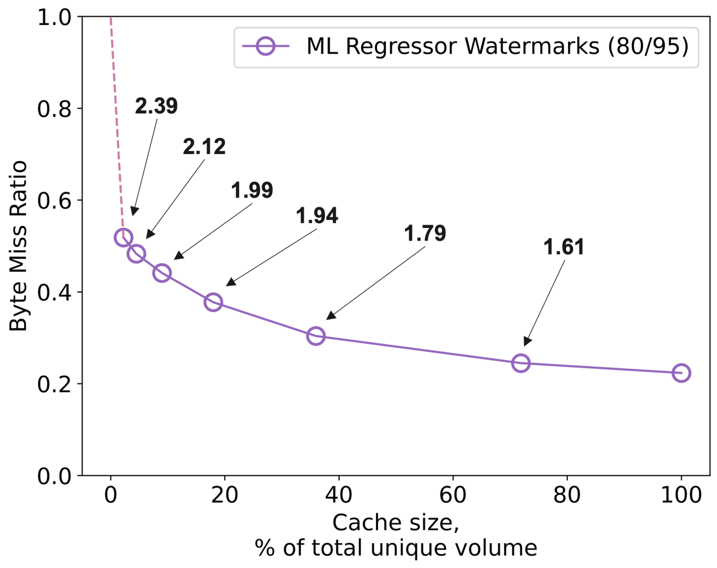


Figure 4.15: Performance evaluation plot of an ML Regressor with watermark thresholds (80/95), with annotated *RMSE* values indicating the prediction error of the ML Regressor for various cache sizes.

of each method.

4.6.1 *LRU* and *LRU Watermarks*

The chosen approach involves implementing the queue as a double-linked list (referred to as dq), which allows for constant time ($O(1)$) insertions and removals, including in the middle of the queue.

An auxiliary map (ma) is used to map file IDs to their respective positions in the queue. This setup guarantees that all operations in the map occur in average constant time ($O(1)$).

The core operations of the system are as follows:

Purging is initiated whenever a new element is inserted into the cache. This process involves removing elements from the cache, starting from the head. Each individual removal and the corresponding update in the map can be performed in constant time $O(1)$. Let l denote the total number of cache purging operations that occur, and let k_i represent the number of elements that need to be purged in the i^{th} purging operation. Here, i ranges from 1 to l, inclusive. N is the trace length.

The total execution time for processing each file access in the trace is given by:

$$T(N) = O(N) + O(k_1 + k_2 + \ldots + k_l) \tag{4.4}$$

Algorithm 4: Cache Management Process

```
1  if element is present in cache then
2      Move element to the head of $dq$ ($O(1)$) ;
3      Update $ma$ ($O(1)$) ;
4  else
5      if element is new then
6          Insert element at the head of $dq$ ($O(1)$) ;
7          Update $ma$ ($O(1)$) ;
8          Trigger purging if necessary ;
9      end
10 end
```

However, considering that the sum of all k values $(k_1, k_2, \ldots, k_l)$ is less than or equal to N, the total execution time simplifies to:

$$T(N) = O(N) + O(N) = O(N) \tag{4.5}$$

When switching to the *LRU Watermarks* implementation, the individual values of k_i may change, but the following remains true:

$$\sum_{i=1}^{l} k_i \leq N \tag{4.6}$$

Consequently, the overall time complexity remains the same, maintaining the efficiency of this approach.

4.6.2 Machine Learning Approach

The time complexity of Random Forest and Decision Tree algorithms primarily depends on two main phases: the training phase and the prediction (inference) phase. Although training generally exhibits higher time complexity, in practical scenarios, the prediction phase often becomes more dominant in terms of integrated costs. This is due to the stability of access patterns over extended periods, such as several months, which leads to a more frequent engagement with the prediction phase.

Decision Tree Time Complexity

Training Time Complexity The training time complexity of a decision tree is primarily determined by the following factors:

- **Number of Data Points (D):** The more data points, the longer it takes to construct the tree.
- **Number of Features (F):** The complexity grows as the number of features increases.
- **Tree Depth ($\log(D)$):** The depth of the tree depends on various factors, including stopping criteria and data complexity, as discussed in Subsection 4.4.3.

Therefore, the training time complexity can be expressed as:

$$\text{Training Time Complexity} = O(D \cdot F \cdot \log(D))$$

Inference (Prediction) Time Complexity A decision tree's prediction is a fast operation because it only needs to traverse a tree from the root to a leaf node based on feature values. The time complexity is:

$$\text{Inference Time Complexity} = O(\log(D))$$

where D is the depth of the tree.

Random Forest Time Complexity

During the training phase of Random Forest, several factors influence the time complexity:

- **Number of Trees (N_{tree}):** The more trees in the forest, the longer the training phase takes.
- **Bootstrapping Size (D_{sample}):** The first step of constructing a Random Forest is typically the Bootstrapping (Random Sampling), where the datasets for training each individual Decision Tree are obtained. The time complexity of bootstrapping depends on the number of trees and the size of the bootstrapping dataset, and equals to $O(N_{tree} \cdot D_{sample})$, as one is sampling D_{sample} data points N_{tree} times.
- **Parallelization Factor (P):** Random Forest can potentially parallelize the training of individual decision trees. Depending on the implementation and available hardware, this can reduce the training time, especially when constructing multiple trees simultaneously. This parallelization factor represents how much parallelism your Spark cluster can achieve. It can depend on the number of worker nodes, CPU cores per node, and the ability to distribute the workload effectively.

Therefore, the overall training time complexity for the Random Forest approach can be expressed as:

$$\text{Training Time Complexity} = O(P \cdot N_{tree} \cdot D_{sample} + P \cdot F \cdot D_{sample} \cdot \log(D_{sample})))$$

Here, $\log(D_{sample})$ represents the average depth of each decision tree.

Prediction (Inference) Time Complexity

The prediction phase of Random Forest is very efficient, with a time complexity of:

$$\text{Inference Time Complexity} = O(P \cdot N_{tree} \cdot \log(D_{sample}))$$

Caching with ML Time Complexity

When the purging process is triggered, predictions (interference) needs to be made for each file present in the cache. Assuming that there are w_i files each time, where i ranges from 1 to l, with l representing the number of cleanup cycles.

During each cleanup cycle, generating predictions incurs the following time complexity:

$$T_{\text{cleanup}}(i) = O(w_i \cdot P \cdot N_{\text{tree}} \cdot \log(D_{\text{sample}}))$$

This complexity arises from the utilisation of the Random Forest model underlying the predictions.

Following prediction generation, the cache files need to be ordered, which takes $O(w_i \cdot \log(w_i))$ time, and a portion of them, denoted as z_i files, must be deleted, which has a time complexity of $O(z_i)$. Since $w_i > z_i$, the result is obtained:

$$T_{\text{purge}}(i) = O(w_i \cdot P \cdot N_{\text{tree}} \cdot \log(D_{\text{sample}}) + w_i \cdot \log(w_i))$$

To determine the overall time complexity for simulating a cache of a specific size, the sum of the time complexities is calculated for each of the l cleanup cycles:

$$T_{\text{total}} = \sum_{i=1}^{l} T_{\text{purge}}(i)$$

Simplifying this formula further becomes challenging due to the variability in the remaining files across different cleanup cycles.

Chapter 5

Conclusions

In this work, the existing scientific workloads were studied, as they are processed at the CERN Data Center, in the context of caching. This included analysis of the access patterns of two three-month traces of ATLAS, one of the two largest LHC experiments. The focus was on the user read accesses for the analysis data, as they correspond to the most unpredictable part of the storage accesses [1]. Furthermore, this research explores in great detail the request rates, time locality, file popularity, and system load depending on the file size, as well as intra-dataset access patterns.

The comprehensive analysis of the EOS disk storage system's data access patterns involved developing a full pipeline for processing EOS report logs. This process encompassed collection, parsing, quality improvement, and aggregation of log data. These results illuminated the variations in data processing among experiments, showcasing the differences in data popularity distributions. This analysis highlights the complexity and diversity inherent in data management for large-scale scientific research.

Trace analysis motivated the selection of existing caching policies (*LRU*, *2-LRU*) that can take advantage of temporal locality characteristics and operate with heterogeneous file sizes. They were compared to new cache policies specifically tailored for dataset-based workloads (*Dataset Evict LRU* and *Dataset LRU*). The evaluation of cache performance with respect to *FMR* and *BMR* led to the proposition of the *PFOO-L.Bytes* algorithm, offering a tighter lower bound for the *Byte Miss Ratio* for any reactive caching policy.

The *Dataset LRU* is greatly dependent on prefetching and offers notable enhancements in both *FMR* and *BMR*. However, concerns arose about the high number of delayed hits. Simulations with varying network throughput values showed that *Dataset LRU* loses its advantage over *LRU* when the throughput is insufficient to sustain higher data volumes.

[1] MC production, Reconstruction, AOD and DAOD generation are centrally organised activities, while analysis is driven by a large number of small teams and individuals

The file popularity predictive model coupled with watermark-based implementation showcases a promising approach for cache eviction. Leveraging the predictive power of tree-based ML models and optimising the cache cleanup process demonstrates the potential to enhance caching efficiency and reduce unnecessary cache evictions. The feature importance distributions of the regression and classification models revealed the significance of file recency and popularity, supporting the efficacy of recency-based eviction policies.

5.1 Future Directions

The research period covered two three-month periods, which captured short-term access patterns but did not represent yearly data access trends in the physics community. Notably, changes in data access patterns occur during the LHC's data-taking periods. In future work, it could be beneficial to extend the analysis to these data as they become available and to explore developing a model for evaluating the global cost of caching.

This study has shed light on the challenges of surpassing *LRU* and integrating ML models effectively into cache eviction algorithms, highlighting promising future directions. To further improve ML-based cache eviction policies, several possible alternative approaches have been identified. Expanding the training dataset could enhance the ML models' performance, allowing them to learn from more diverse data. Exploring different combinations of ML models and optimising their hyperparameters remains an area of opportunity.

Bibliography

[1] CERN: European Organization for Nuclear Research. https://home.cern/. Accessed: 2024-01-11.

[2] Large Hadron Collider - CERN. https://home.cern/science/accelerators/large-hadron-collider. Accessed: 2024-01-11.

[3] ATLAS Collaboration, G Aad, E Abat, J Abdallah, AA Abdelalim, A Abdesselam, O Abdinov, BA Abi, M Abolins, H Abramowicz, et al. The ATLAS experiment at the CERN large hadron collider. *Journal of Instrumentation*, 3(08):S08003, 2008.

[4] CMS Collaboration, S Chatrchyan, G Hmayakyan, V Khachatryan, AM Sirunyan, W Adam, T Bauer, T Bergauer, H Bergauer, M Dragicevic, et al. The CMS experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08004, 2008.

[5] Kenneth Aamodt, A Abrahantes Quintana, R Achenbach, S Acounis, D Adamová, C Adler, M Aggarwal, F Agnese, G Aglieri Rinella, Z Ahammed, et al. The ALICE experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08002, 2008.

[6] TL Collaboration, A Augusto Alves, LM Andrade Filho, AF Barbosa, I Bediaga, G Cernicchiaro, G Guerrer, HP Lima, AA Machado, J Magnin, et al. The LHCb detector at the LHC. *Journal of instrumentation*, 3(08):S08005–S08005, 2008.

[7] Worldwide LHC Computing Grid (WLCG) - CERN. https://wlcg.web.cern.ch/. Accessed: 2024-01-11.

[8] ATLAS Collaboration. ATLAS Computing: Technical Design Report. ATLAS-TDR-17; CERN-LHCC-2005-022, CERN, 2005.

[9] Ian Bird, F Carminati, R Mount, B Panzer-Steindel, J Harvey, Ian Fisk, B Kersevan, P Clarke, M Girone, P Buncic, et al. Update of the Computing Models of the WLCG and the LHC Experiments. CERN-LHCC-2014-014; LCG-TDR-002, CERN, 2014.

[10] Sea Agostinelli, John Allison, K al Amako, John Apostolakis, H Araujo, Pedro Arce, Makoto Asai, D Axen, Swagato Banerjee, GJNI Barrand, et al. GEANT4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003.

[11] P Calafiura, M Marino, C Leggett, W Lavrijsen, and D Quarrie. The Athena control framework in production, new developments and lessons learned. 2005.

[12] ROOT—An object oriented data analysis framework,.

[13] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[14] CERN. The CERN Data Centre. https://home.cern/science/computing/data-centre, 2023. Accessed: 2023-12-16.

[15] CERN. LHCOPN. https://lhcopn.web.cern.ch/. Accessed: 2024-01-11.

[16] CERN. LHCONE. https://lhcone.web.cern.ch/. Accessed: 2024-01-11.

[17] Tadashi Maeno. PanDA: distributed production and distributed analysis system for ATLAS. In *Journal of Physics: Conference Series*, volume 119, page 062036. IOP Publishing, 2008.

[18] Daniele Spiga, Stefano Lacaprara, W Bacchi, Mattia Cinquilli, Giuseppe Codispoti, Marco Corvo, A Dorigo, Alessandra Fanfani, Federica Fanzago, Fabio Farina, et al. The CMS remote analysis builder (CRAB). In *High Performance Computing–HiPC 2007: 14th International Conference, Goa, India, December 18-21, 2007. Proceedings 14*, pages 580–586. Springer, 2007.

[19] CERN. Rucio - Scientific Data Management. https://lhcone.web.cern.ch/. Accessed: 2024-01-11.

[20] AC Smith and A Tsaregorodtsev. DIRAC: reliable data management for LHCb. In *Journal of Physics: Conference Series*, volume 119, page 062045. IOP Publishing, 2008.

[21] CERN. VOMS Home. https://italiangrid.github.io/voms/. Accessed: 2024-01-11.

[22] Julia Andreeva, Alexey Anisenkov, Alessandro Di Girolamo, Alessandra Forti, Stephen Jones, Balazs Konya, Andrew McNab, and Panos Paparrigopoulos. Evolution of the WLCG Information Infrastructure. In *EPJ Web of Conferences*, volume 245, page 03029. EDP Sciences, 2020.

[23] Gilles Mathieu, Andrew Richards, John Gordon, Cristina Del Cano Novales, Peter Colclough, and Matthew Viljoen. GOCDB, a topology repository for a worldwide grid infrastructure. In *Journal of Physics: Conference Series*, volume 219, page 062021. IOP Publishing, 2010.

[24] CERN. FTS - File Transfer Service. `https://fts.web.cern.ch/fts/`. Accessed: 2024-01-11.

[25] CERN. WLCG VObox Deployment Documentation. `https://twiki.cern.ch/twiki/bin/view/LCG/WLCGvoboxDeployment`. Accessed: 2024-01-11.

[26] CERN. CernVM File System. `https://cernvm.cern.ch/fs/`. Accessed: 2024-01-11.

[27] Jana Schaarschmidt, Johannes Elmsheuser, Lukas Alexander Heinrich, Nils Erik Krumnack, James Catmore, Alaettin Serhan Mete, and Nurcan Ozturk. PHYSLITE-A new reduced common data format for ATLAS. Technical report, ATL-COM-SOFT-2023-105, 2023.

[28] A Joachim Peters, Elvin Alin Sindrilaru, and Geoffrey Adde. EOS as the present and future solution for data storage at CERN. In *Journal of Physics: Conference Series*, volume 664, page 042042. IOP Publishing, 2015.

[29] Paolo Calafiura, James Catmore, Davide Costanzo, and Alessandro Di Girolamo. ATLAS HL-LHC computing conceptual design report. Technical report, 2020.

[30] ATLAS collaboration and others. ATLAS software and computing HL-LHC roadmap. Technical report, 2022.

[31] Martin Barisits, Mikhail Borodin, Alessandro Di Girolamo, Johannes Elmsheuser, Dmitry Golubkov, Alexei Klimentov, Mario Lassnig, Tadashi Maeno, Rodney Walker, and Xin Zhao. ATLAS Data Carousel. In *EPJ Web of Conferences*, volume 245, page 04035. EDP Sciences, 2020.

[32] Mikhail Borodin, Alessandro Di Girolamo, Edward Karavakis, Alexei Klimentov, Tatiana Korchuganova, Mario Lassnig, Tadashi Maeno, Siarhei Padolski, and Xin

Zhao. The ATLAS Data Carousel Project Status. In *EPJ Web of Conferences*, volume 251, page 02006. EDP Sciences, 2021.

[33] Ian Bird, Simone Campana, Maria Girone, Xavier Espinal, Gavin McCance, and Jaroslava Schovancová. Architecture and prototype of a WLCG data lake for HL-LHC. In *EPJ Web of Conferences*, volume 214, page 04024. EDP Sciences, 2019.

[34] I Kadochnikov, I Bird, G McCance, J Schovancova, M Girone, S Campana, and XE Currul. WLCG data lake prototype for HL-LHC. Advisory committee, 127 (2018).

[35] Maria Grigorieva, Eugeny Tretyakov, Alexei Klimentov, Dmitry Golubkov, Tatiana Korchuganova, Aleksandr Alekseev, Alexey Artamonov, and Timofei Galkin. High Energy Physics Data Popularity: ATLAS Datasets Popularity Case Study. In *2020 Ivannikov Memorial Workshop (IVMEM)*, pages 22–28. IEEE, 2020.

[36] Thomas Beermann, Olga Chuchuk, Alessandro Di Girolamo, Maria Grigorieva, Alexei Klimentov, Mario Lassnig, Markus Schulz, Andrea Sciaba, and Eugeny Tretyakov. Methods of Data Popularity Evaluation in the ATLAS Experiment at the LHC. In *EPJ Web of Conferences*, volume 251, page 02013. EDP Sciences, 2021.

[37] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[38] Michele Garetto, Emilio Leonardi, and Valentina Martina. A unified approach to the performance analysis of caching systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 1(3):1–28, 2016.

[39] Alireza Montazeri, Nicholas R Beaton, and Dwight Makaroff. LRU-2 vs 2-LRU: An Analytical Study. In *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, pages 571–579. IEEE, 2018.

[40] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.

[41] Nimrod Megiddo and Dharmendra S Modha. {ARC}: A {Self-Tuning}, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.

[42] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S Berger. Caching with delayed hits. pages 495–513, 2020.

[43] Huichen Dai, Bin Liu, Haowei Yuan, Patrick Crowley, and Jianyuan Lu. Analysis of tandem PIT and CS with non-zero download delay. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[44] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 483–498, 2017.

[45] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Hewlett-Packard Laboratories, 1998.

[46] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[47] Giovanni Neglia, Damiano Carra, and Pietro Michiardi. Cache policies for linear utility maximization. *IEEE/ACM Transactions on Networking*, 26(1):302–313, 2018.

[48] Giuseppe Rossini, Dario Rossi, Michele Garetto, and Emilio Leonardi. Multi-terabyte and multi-gbps information centric routers. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 181–189. IEEE, 2014.

[49] Ernst W Biersack, Alain Jean-Marie, and Philippe Nain. Open-loop video distribution with support of VCR functionality. *Performance Evaluation*, 49(1-4):411–427, 2002.

[50] Lei Lei, Lei You, Gaoyang Dai, Thang Xuan Vu, Di Yuan, and Symeon Chatzinotas. A deep learning approach for optimizing content delivering in cache-enabled hetnet. In *2017 international symposium on wireless communication systems (ISWCS)*, pages 449–453. IEEE, 2017.

[51] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 48–53, 2018.

[52] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *Journal of the ACM (JACM)*, 68(4):1–25, 2021.

[53] Herodotos Herodotou. AutoCache: Employing machine learning to automate caching in distributed file systems. In *2019 IEEE 35th international conference on data engineering workshops (ICDEW)*, pages 133–139. IEEE, 2019.

[54] Alireza Sadeghi, Gang Wang, and Georgios B Giannakis. Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. *IEEE Transactions on Cognitive Communications and Networking*, 5(4):1024–1033, 2019.

[55] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. *ACM SIGARCH Computer Architecture News*, 44(3):78–89, 2016.

[56] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425, 2019.

[57] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pages 6237–6247. PMLR, 2020.

[58] Derek Weitzel, Brian Bockelman, Duncan A Brown, Peter Couvares, Frank Würthwein, and Edgar Fajardo Hernandez. Data Access for LIGO on the OSG. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, pages 1–6, 2017.

[59] Teng Li, Robert Currie, and Andrew Washbrook. A data caching model for Tier 2 WLCG computing centres using XCache. In *EPJ Web of Conferences*, volume 214, page 04047. EDP Sciences, 2019.

[60] Aleksandr Alekseev, Stephane Jezequel, Andrey Kiryanov, Alexei Klimentov, Tatiana Korchuganova, Valery Mitsyn, Danila Oleynik, Serge Smirnov, and Andrey Zarochentsev. Evaluation of the Impact of Various Local Data Caching Configurations on Tier2/Tier3 WLCG Sites. In *CEUR Workshop Proceedings*, volume 2679, pages 1–10. RWTH Aahen University, 2020.

[61] LAT Bauerdick, K Bloom, B Bockelman, DC Bradley, S Dasu, JM Dost, I Sfiligoi, A Tadel, M Tadel, F Wuerthwein, et al. XRootd, disk-based, caching proxy for optimization of data access, data placement and data replication. *Journal of Physics: Conference Series*, 513(4):042044, 2014.

[62] Gaëtan Heidsieck, Daniel De Oliveira, Esther Pacitti, Christophe Pradal, Francois Tardieu, and Patrick Valduriez. Efficient execution of scientific workflows in the cloud through adaptive caching. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLIV*, pages 41–66. Springer, 2020.

[63] Olga Chuchuk and Dirk Duellmann. Access Pattern Analysis in the EOS Storage System at CERN. In *CEUR Workshop Proceedings*, pages 22–31, 2020.

[64] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[65] Marek Chrobak, Gerhard J Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[66] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181, 2013.

[67] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–38, 2018.

[68] Nitish K Panigrahy, Philippe Nain, Giovanni Neglia, and Don Towsley. A New Upper Bound on Cache Hit Probability for Non-anticipative Caching Policies. *ACM SIGMETRICS Performance Evaluation Review*, 48(3):138–143, 2021.

[69] Alberto Aimar, Asier Aguado Corman, Pedro Andrade, Javier Delgado Fernandez, Borja Garrido Bear, Edward Karavakis, Dominik Marek Kulikowski, and Luca Magnoni. MONIT: monitoring the CERN data centres and the WLCG infrastructure. In *EPJ Web of Conferences*, volume 214, page 08031. EDP Sciences, 2019.

[70] Martin Barisits, Thomas Beermann, Frank Berghaus, Brian Bockelman, Joaquin Bogado, David Cameron, Dimitrios Christidis, Diego Ciangottini, Gancho Dimitrov, Markus Elsing, et al. Rucio: Scientific data management. *Computing and Software for Big Science*, 3(1):1–19, 2019.

[71] Grafana Labs. Grafana - the open observability platform. https://grafana.com/oss/grafana/, 2024. Accessed: 2024-01-11.

[72] Felipe Olmos and Bruno Kauffmann. An inverse problem approach for content popularity estimation. *arXiv preprint arXiv:1510.07480*, 2015.

[73] Wes McKinney and the pandas development team. pandas: powerful Python data analysis toolkit. Version 1.5.0. https://pandas.pydata.org/, 2023. Accessed: 2024-01-11.

[74] Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981.

[75] Damiano Carra and Giovanni Neglia. Efficient miss ratio curve computation for heterogeneous content popularity. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 741–751, 2020.

[76] Guilherme Iecker Ricardo, Alina Tuholukova, Giovanni Neglia, and Thrasyvoulos Spyropoulos. Caching policies for delay minimization in small cell networks with coordinated multi-point joint transmissions. *IEEE/ACM Transactions on Networking*, 29(3):1105–1115, 2021.

[77] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[78] Martin Sewell. Ensemble learning. *RN*, 11(02):1–34, 2008.

[79] Apache Spark Documentation. pyspark.ml.regression. https://spark.apache.org/docs/latest/ml-classification-regression.html#regression, 2024. Accessed: 2024-01-11.

[80] RandomForestRegressor - PySpark 3.1.1 documentation. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.regression.RandomForestRegressor.html. Accessed: 2023-12-21.