Boosting Hive Efficiency: A Novel Dual-Process Architecture for Asynchronous and Parallel Data Loading

by

Mohammad Ashekur Rahman
21166025

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
M.Sc. in Computer Science and Engineering

Department of Computer Science and Engineering
Brac University
October 2024

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Mohammad Ashekur Rahman

21166025

Approval

The thesis titled "Boosting Hive Efficiency: A Novel Dual-Process Architecture for Asynchronous and Parallel Data Loading" submitted by

1. Mohammad Ashekur Rahman (21166025)

Of Fall, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of M.Sc. in Computer Science and Engineering on October 16, 2024.

Examining Committee:

External Examiner:
(Member)

M.M.A. Hashem, PhD

Professor
Department of Computer Science and Engineering
Khulna University of Engineering & Technology

Internal Examiner:
(Member)

Muhammad Iqbal Hossain, PhD

Associate Professor
Department of Computer Science and Engineering
Brac University

Supervisor:
(Member)

Farig Yousuf Sadeque, PhD

Associate Professor
Department of Computer Science and Engineering
Brac University

Program Coordinator:
(Member)

Md Sadek Ferdous, PhD

Associate Professor
Department
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD

Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Ethics Statement

Throughout the research process, I have made extensive use of various scholarly sources, including journals, conference publications, and reputable websites, in compliance with academic standards. I also utilized assistance from AI-based tools like ChatGPT and Copilot to aid in the development of ideas and in refining the structure of the report. All sources and tools used have been duly acknowledged, and no part of this work has been copied or plagiarized without proper citation.

Abstract

The efficiency of data loading processes in Hive, a critical component of modern big data ecosystems, is often hindered by sequential bottlenecks that limit overall performance. This thesis proposes a novel asynchronous and parallel data loading architecture designed to address these challenges, enhancing Hive's data ingestion capabilities. The architecture comprises two distinct processes: the Landing Batch Process, which manages data loading into the Hadoop Distributed File System (HDFS), and the Staging Batch Process, responsible for loading data into Hive tables. By operating these processes asynchronously and in parallel, the proposed design significantly accelerates data handling.

Experimental evaluations compared the performance of the proposed architecture in scenarios without parallelism and with two parallel processes against the traditional sequential approach. Three diverse datasets—NOAA weather data, Threat data, and Stock market data—were tested to assess the scalability and robustness of the solution. The results revealed substantial performance improvements across all datasets. The NOAA dataset exhibited a reduction in total processing time of 42%, the Threat dataset achieved a 42.5% reduction, and the Stock dataset showed the greatest improvement, with a 43.42% decrease in total processing time. Notably, parallel processing reduced the landing time from 451 seconds to 402.66 seconds for the NOAA dataset, from 861 seconds to 763 seconds for Threat data, and from 2,643 seconds to 2,342 seconds for Stock data. Additionally, the average landing iteration time was significantly reduced across the datasets, further underscoring the efficiency gains of parallel execution.

These findings demonstrate the broad applicability and efficiency of the proposed architecture, making it a powerful tool for overcoming the traditional limitations of Hive's data loading processes in high-volume environments. This thesis concludes that the asynchronous and parallel approach offers a significant advancement in data loading efficiency, making it a viable solution for high-volume data environments. Future research will explore further optimization of the staging process, scalability analysis with additional parallel processes, and integration with real-time data frameworks, aiming to establish a robust and scalable architecture for big data applications in Hive and beyond.

Keywords: Data Loading; Hive; Parallel Processing; Asynchronous Processing; Multiprocessing; Big Data; Data Warehouse; Data Lake; Hadoop

Dedication

This thesis is dedicated to my loving parents, whose unwavering support has always been my strength; to my beloved wife, whose patience and encouragement kept me motivated throughout this journey; and to my esteemed supervisor, **Dr. Farig Yousuf Sadeque**, whose guidance and wisdom were instrumental in the successful completion of this work.

Acknowledgement

First and foremost, I would like to express my deepest gratitude to **Allah** for giving me the strength and determination to complete this thesis.

I would like to extend my sincere appreciation to my supervisor, **Dr. Farig Yousuf Sadeque**, for his steadfast support, guidance, and invaluable advice. His door was always open, and whenever I sought his help, he welcomed me with a smile and provided the assistance I needed to stay on track.

A special thank you goes to my beloved parents and wife, whose constant love, understanding, and encouragement fueled my drive to succeed in this endeavor.

Finally, I would like to thank my colleagues and friends for their support throughout this journey. Their encouragement made the challenging moments more bearable, and their understanding during my busy thesis days was greatly appreciated.

Table of Contents

Declaration i

Approval ii

Ethics Statement iv

Abstract v

Dedication vi

Acknowledgment vii

Table of Contents viii

List of Figures xi

List of Tables xii

Nomenclature xiii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Gap and Contribution . . . 4
- 1.3 Significance of the Proposed Solution . . . 4
- 1.4 Research Questions . . . 5
- 1.5 Methodological Approach . . . 5
- 1.6 Outline of the Thesis . . . 5

2 Literature Review 7
- 2.1 Literature Review . . . 7

3 Background Study 14
- 3.1 Apache Hive . . . 14
  - 3.1.1 Key Features of Apache Hive . . . 14
  - 3.1.2 Hive Architecture . . . 15
  - 3.1.3 Hive Table Types . . . 15
  - 3.1.4 Hive Data Model . . . 18
  - 3.1.5 Common Use Cases . . . 19
  - 3.1.6 Advantages and Limitations . . . 19
- 3.2 Apache Hive Data Loading Architecture . . . 20

3.2.1 Data Sources . . . 20
3.2.2 Key Components of Hive Data Loading . . . 20
3.2.3 Data Loading Methods in Hive . . . 21
3.2.4 Partitioning and Bucketing in Hive . . . 22
3.2.5 File Formats and Compression . . . 23
3.2.6 Schema Evolution and Data Management . . . 26
3.2.7 Best Practices for Data Loading in Hive . . . 26
3.2.8 Data Loading Workflow in Hive . . . 26
3.3 Traditional Hive Data Ingestion Architecture (Any File Type) . . . 27
3.4 Choosing Right File Type . . . 29
3.4.1 Challenges of Storing Any Type of File . . . 29
3.4.2 Why ORC is Best for Transactional Data, Performance, and Size . . . 29
3.5 Traditional Hive Data Ingestion Architecture (Any File type to ORC) 30
3.5.1 Landing Batch Process . . . 31
3.5.2 Staging Batch Process . . . 31
3.5.3 Benefits of This Architecture . . . 32
3.6 Sequential Process Bottlenecks . . . 33
3.6.1 Why Sequential Processing is Slow . . . 33
3.6.2 Potential Improvements . . . 33

4 Research Methodology 35
4.1 Proposed Architecture . . . 35
4.1.1 Two Asynchronous Processes . . . 35
4.1.2 Master Landing & Staging Batch Process . . . 37
4.1.3 Child Landing Batch Process . . . 38
4.1.4 Child Staging Batch Process . . . 41
4.1.5 Required Configuration for Data Loading . . . 42
4.1.6 One Single INSERT Query to Start Data Loading . . . 45
4.2 Experimental Environment and Assumptions . . . 45

5 Implementation 46
5.1 Dataset . . . 46
5.1.1 Dataset Descriptions . . . 46
5.1.2 Dataset Composition . . . 47
5.1.3 Data Format and Characteristics . . . 49
5.1.4 Relevance to the Study . . . 49
5.1.5 Challenges and Solutions . . . 51
5.2 Environment Setup . . . 54
5.2.1 Amazon EMR . . . 54
5.2.2 EMR Cluster Configuration . . . 54
5.2.3 Integration with Apache Hive and Apache Spark . . . 55
5.2.4 Gateway/Staging/Edge Node . . . 55
5.3 Data Preprocessing . . . 56
5.4 Data Loading . . . 58
5.4.1 Traditional Architecture . . . 58
5.4.2 Proposed Architecture . . . 59

6 Result and Analysis **61**
6.1 Data Size Analysis . . . 61
6.1.1 Traditional Data Loading with Raw Files . . . 61
6.1.2 Optimized Data Storage with ORC Format . . . 62
6.1.3 Comparison of Storage Reduction . . . 62
6.1.4 Implications for Data Warehousing . . . 63
6.2 Landing Batch Process with No Parallel . . . 63
6.3 Landing Batch Process with 2 Parallel . . . 65
6.4 Analysis of Both Landing Batch Process . . . 67
6.5 Staging Batch Process . . . 68
6.6 Performance Comparison . . . 70
6.6.1 Analysis . . . 70
6.6.2 Key Observations: . . . 71
6.6.3 Detailed Dataset-Specific Performance Comparison . . . 72
6.6.4 Overall Improvement Comparison . . . 73

7 Conclusion and Future Work **74**
7.1 Conclusion . . . 74
7.2 Limitations . . . 75
7.3 Future Work . . . 75

Bibliography **78**

List of Figures

3.1 Hive Architecture 15
3.2 Hive Data Model 19
3.3 Traditional Hive Data Ingestion Architecture (Any File Type) 27
3.4 Traditional Hive Data Ingestion Architecture (Any File type to ORC) 30

4.1 Proposed Architecture – Two Asynchronous Processes 35
4.2 Master Landing & Staging Batch Process 37
4.3 Child Landing Batch Process 39
4.4 Child Staging Batch Process 41

6.1 Normal Distribution of Landing Batch Process with No Parallel for NOAA Dataset 64
6.2 Normal Distribution of Landing Batch Process with 2 Parallel for NOAA Dataset 66
6.3 Normal Distribution of Staging Batch Process 69

List of Tables

3.1 Comparison of Hive Managed and External Tables 18
3.2 Comparison of File Formats and Properties 23

6.1 Comparison of File Formats and Storage Sizes Across Datasets 62
6.2 Storage Reduction with ORC Format Across Datasets 62
6.3 Landing Batch Process with No Parallel - Normal Distribution Values for NOAA Dataset 63
6.4 Landing Batch Process with 2 Parallel - Normal Distribution Values for NOAA Dataset 66
6.5 Comparison of Non-Parallel and 2 Parallel Landing Batch Process for All Datasets 68
6.6 Staging Batch Process - Normal Distribution Values (NOAA Dataset) 69
6.7 Performance Comparison of Traditional and Proposed Architectures for NOAA Dataset 72
6.8 Performance Comparison of Traditional and Proposed Architectures for Threat Data 72
6.9 Performance Comparison of Traditional and Proposed Architectures for Stock Data 72
6.10 Comparison of Landing and Total Improvements Across Different Sources 73

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

$ACID$ Atomicity, Consistency, Isolation, and Durability

ETL Extract, transform, and load

$HADOOP$ High Availability Distributed Object Oriented Platform

$HDFS$ Hadoop Distributed File System

$HPCC$ High Performance Computing Cluster

$JDBC$ Java Database Connectivity

$LLAP$ Live Long and Process

MPP Massively Parallel Processing

$ODBC$ Open Database Connectivity

ORC Optimized Row Columnar

SQL Structured Query Language

Chapter 1

Introduction

1.1 Motivation

Data analytics plays a crucial role in modern organizations by providing insights that inform strategic decisions. Central to data analytics is the process of data integration, which consolidates historical data from diverse sources into a centralized data warehouse. The Extract Transform Load (ETL) process stands out as a popular and efficient method for achieving this integration. ETL has been widely adopted across domains such as finance, healthcare, and telecommunications. As the utilization of ETL continues to expand, gaining insights into its research advancements and practical applications becomes increasingly important [19].

Extraction, Transformation, and Loading (ETL) processes are vital for the backend operations of a data warehouse architecture. They involve extracting data from various source datastores, transforming the data for homogenization and cleansing, and then loading it into a central data warehouse. ETL processes have been integral to database technology since its inception. The ETL workflow includes three main steps: extraction, transformation, and loading. These processes are crucial in modern corporate environments, necessitating a dedicated team for designing and maintaining ETL functionalities. There is a need for a unified algebra or a declarative language for formally describing ETL processes, optimizing the entire ETL workflow, and expanding ETL capabilities beyond traditional data warehouse environments [3].

Big data encompasses a wide range of content, scope, methods, advantages, challenges, and privacy concerns. It is not solely about managing large volumes of data but also about extracting meaningful insights and value from it. Essential analysis methods include distributed programming, pattern recognition, data mining, natural language processing, sentiment analysis, statistical and visual analysis, and human-computer interaction [7]. Key technologies for big data analysis include Hadoop and High-Performance Computing Cluster (HPCC). Hadoop, a Java-based framework, incorporates a distributed file system, analytics and data storage platforms, and a layer for managing parallel computation, workflow, and configuration administration. In contrast, HPCC is a distributed data-intensive open-source computing platform offering comprehensive big data workflow management services. Privacy and security issues are significant in the realm of big data. Despite its numerous ben-

efits, big data presents challenges such as data growth, infrastructure, governance, integration, velocity, variety, compliance/regulation, and visualization. Addressing these challenges requires continuous research, improvement, and development in the field. Moreover, managers and analysts must develop a deep understanding of big data applications to leverage its full potential effectively.

A data lake implementation necessitates the integration of data from various sources, including existing operational systems, data warehouses, and new data streams. Given Hadoop's capabilities as a general-purpose, large-scale distributed processing platform, it is particularly well-suited for this task. The initial step in enabling analytics within this data lake is the data loading process. Traditionally, data was extracted from operational systems and loaded into a data warehouse in batch form. Historically, no single tool in the Hadoop ecosystem could handle data loading from all systems and formats. Instead, a variety of tools were developed, each optimized for specific systems and data formats. This approach can lead to complexity when loading data from multiple sources in different formats. The complexity is further compounded by factors such as the frequency of data loading from source systems, which can influence the choice of the most appropriate tool. Regardless of the source type, data structure, and tools used for loading, all data in a Hadoop-based platform is ultimately stored in HDFS. Since Hive serves as the SQL layer on Hadoop, data must be loaded into HDFS before it can be queried through Hive [14].

Hive is an open-source data warehousing solution built on top of Hadoop, designed to manage petabyte-scale data sets using commodity hardware. It provides a SQL-like declarative language known as HiveQL, which translates into MapReduce jobs executed by Hadoop. HiveQL supports complex data types and allows users to incorporate custom MapReduce scripts. The architecture of Hive comprises several key components:

- **Metastore:** Stores metadata about tables, partitions, and other data warehouse objects.
- **Compiler:** Generates execution plans for queries.
- **Execution Engine:** Executes the compiled queries by interacting with Hadoop.
- **JDBC/ODBC Server:** Enables integration with other applications.

Hive's storage model organizes data into tables, partitions, and buckets, which are stored in HDFS directories. It supports multiple file formats and SerDes (serialization/deserialization) for efficient data processing. The query compiler in Hive optimizes HiveQL statements using techniques such as column pruning, predicate pushdown, partition pruning, and join reordering. It also addresses data skew in GROUP BY operations and uses hash-based partial aggregations in mappers to enhance performance [4].

Apache Hive has evolved from a simple ETL tool to a comprehensive, enterprise-grade data warehouse solution. It is designed to manage substantial big data workloads, integrating traditional MPP (Massively Parallel Processing) techniques with

modern big data and cloud computing concepts. Hive supports standard SQL operations and ACID transactions, ensuring data integrity through Snapshot Isolation facilitated by a transaction manager built on the Hive Metastore. To enhance query performance, Hive leverages Apache Calcite for rule-based and cost-based optimizations, encompassing strategies like join reordering and predicate simplification. Hive incorporates advanced features such as query reoptimization, results caching, materialized views with automatic query rewriting, and shared work optimization to reuse execution plans efficiently. Additionally, Hive serves as a mediator for querying multiple data management systems, utilizing Apache Calcite for optimization and federated query capabilities. Performance evaluations using industry-standard benchmarks illustrate substantial enhancements in Hive's query execution speed and SQL functionality over successive iterations. Future research directions for Hive include further refining optimization techniques, implementing multi-statement transactions, and facilitating seamless deployment in cloud environments through containerization strategies. Hive is adaptable and relevant in the continuously evolving landscape of data analytics [16].

Significant improvements in Hive include the introduction of an efficient file format (ORC File), an updated query planner, and a new vectorized query execution engine that enhances performance by better utilizing modern CPUs [10]. These advancements are demonstrated through experiments using benchmarks like SS-DB, TPC-H, and TPC-DS, showing reduced data sizes, faster query execution times, and more efficient CPU usage. The key improvements include:

1. **Optimized Record Columnar File (ORC File):** Introduced for efficient storage, ORC files use lightweight compression and columnar storage. They support predicate pushdown and include lightweight indexes.

2. **Updated Query Planner:** Enhancements optimize query execution plans by considering statistics, partition pruning, and cost-based optimization. Dynamic partition pruning skips unnecessary partitions during execution.

3. **Vectorized Query Execution:** Processing data in batches (vectors) improves performance. SIMD instructions speed up operations, reducing CPU overhead.

4. **LLAP (Live Long and Process):** In-memory caching layer for Hive, LLAP keeps frequently accessed data in memory, reducing disk I/O and providing low-latency query responses.

5. **ACID Transactions:** Introduced for data consistency and reliability, ACID tables support insert, update, and delete operations in Hive.

1.2 Research Gap and Contribution

Despite significant advancements in Hive and related ETL technologies, the traditional sequential data loading methods remain a major bottleneck in the data integration pipeline, especially when processing large and diverse datasets. Current approaches often struggle to fully exploit parallelism, resulting in inefficient resource utilization, increased loading times, and a failure to meet the demands of modern data-intensive applications. Additionally, existing data loading techniques in Hadoop and Hive environments often rely on a fragmented toolset, which increases complexity when dealing with data from multiple sources and formats. These limitations underscore the need for a more robust and efficient architecture that can address these challenges effectively.

This research proposes an innovative asynchronous and parallel data loading architecture for Hive, which decouples the data ingestion and transformation processes into two distinct stages: the Landing Batch Process and the Staging Batch Process. By allowing these processes to operate independently and concurrently, the proposed architecture maximizes the utilization of system resources and significantly reduces overall data processing times. This approach not only enhances the performance of data loading into Hive but also sets a new benchmark for high-throughput, scalable data warehousing solutions.

1.3 Significance of the Proposed Solution

The proposed asynchronous and parallel data loading architecture holds substantial potential for transforming how data is ingested and processed in big data environments. By optimizing the data loading workflow, the architecture ensures faster data availability for downstream analytics, enabling organizations to derive timely insights and make data-driven decisions more effectively. This architecture is particularly relevant in sectors like finance, healthcare, telecommunications, and e-commerce, where large volumes of data must be processed quickly to maintain a competitive edge.

Moreover, the proposed approach offers a flexible and scalable solution that can adapt to varying data volumes and formats, reducing dependency on traditional, tool-specific data loading methods. The architecture's ability to operate asynchronously means it can better accommodate real-time data integration needs, making it highly suitable for applications involving streaming data and continuous updates.

1.4 Research Questions

This study seeks to address the following key research questions:

1. How does the proposed asynchronous and parallel data loading architecture improve the efficiency of data ingestion in Hive compared to traditional sequential methods?

2. What are the performance implications of implementing parallel processing in the Landing Batch Process, and how does this impact the overall system throughput?

3. Can the decoupling of data ingestion and transformation processes reduce bottlenecks and enhance scalability in large-scale data environments?

1.5 Methodological Approach

To evaluate the effectiveness of the proposed architecture, a series of experiments were conducted comparing the performance of traditional, non-parallel, and parallel data loading approaches in Hive. The experimental setup involved measuring key performance metrics such as landing time, staging time, and total processing time across different scenarios. The analysis focused on quantifying the improvements in data loading efficiency and assessing the impact of parallelism on system performance.

In this thesis, we focus primarily on optimizing data loading processes in Hive, considering the ideal case where variables such as network congestion and bandwidth are not limiting factors. However, future research should consider these variables to explore their impact on the proposed architecture under more realistic conditions.

1.6 Outline of the Thesis

This thesis is structured as follows:

Chapter 2 - Literature Review: This chapter provides a comprehensive overview of existing ETL processes, data loading techniques in Hive, and related research on asynchronous and parallel data architectures. It sets the foundation by exploring current methods and identifying gaps that this research aims to address.

Chapter 3 - Background Study: This chapter covers the background necessary for understanding the proposed architecture, including a detailed description of Hive, its architecture, and its role in big data environments. It also discusses the challenges faced with traditional data loading methods and the need for innovative solutions.

Chapter 4 - Research Methodology: This chapter outlines the research design and methodology used to evaluate the proposed architecture. It provides a detailed

explanation of the design and implementation of the proposed asynchronous and parallel data loading architecture. It covers the architecture's components, workflows, and how it integrates within the Hive ecosystem. The methodology for comparing the proposed architecture with traditional approaches is also detailed.

Chapter 5 - Implementation: This chapter describes the experimental setup, including datasets, testing environments, and performance metrics.

Chapter 6 - Results and Analysis: This chapter presents the results of the experiments conducted, including a comparative analysis of the performance of the proposed architecture versus traditional methods. It highlights the efficiency gains and discusses observed patterns and implications of the findings.

Chapter 7 - Conclusion and Future Work: This chapter summarizes the key contributions of the research, addresses the limitations of the study, and provides recommendations for future research and enhancements. It reflects on how the proposed architecture can be further optimized and its potential impact on data processing in big data environments.

Chapter 2

Literature Review

2.1 Literature Review

Efficient data loading is essential for achieving optimal performance in large-scale distributed deep neural network (DNN) training. In [18], the authors identify performance and scalability issues in current data loading implementations and propose optimizations that leverage CPU resources to enhance data loader design. The study employs an analytical model to characterize the impact of data loading on overall training time and to establish performance trends when scaling up distributed training. It reveals that the I/O rate constrains the scalability of distributed training, leading to the development of a locality-aware data loading method. This method minimizes data loading communication volume by utilizing software caches. The proposed optimizations, encompassing data loader enhancements, a locality-aware data loading method, and an analytical model, are evaluated through experiments. These experiments demonstrate a more than 30x speedup in data loading when using 256 nodes with 1,024 learners, along with a 92% improvement in per-epoch training cost for Imagenet-1K classification over standard distributed training implementations. The paper addresses the challenges and solutions for efficient data loading in large-scale distributed DNN training on high-performance computing (HPC) systems. It highlights the limitations of existing data loader designs and introduces performance optimizations, including a locality-aware data loading method that employs caches to reduce data loading volume and bandwidth requirements. Experimental results showcase significant speedups in data loading and training times across various datasets, including Imagenet-1K, UCF101, and a large molecular dynamics dataset (MuMMI), while maintaining comparable model accuracy. The proposed optimizations enable distributed DNN training to effectively scale to larger HPC systems, thereby enhancing the efficiency and performance of large-scale DNN training.

The methodology proposed in [2] optimizes data warehouse loading procedures to enable real-time data warehousing. It introduces a solution for continuous data integration with minimal impact on query execution time, achieved through schema adaptation. This adaptation involves creating replicas of original schema tables without constraints, thereby facilitating faster data insertion. The ETL loading process is streamlined by focusing on inserting new rows into these temporary tables. The methodology also includes OLAP query adaptation and periodic packing

and reoptimization of the data warehouse database to maintain performance. The objective is to provide up-to-date decision-making information by efficiently integrating the latest data from OLTP systems into the data warehouse, thus enabling real-time data warehousing. It emphasizes the importance of minimizing query execution impact on users by replicating data structures and adapting query instructions. It involves updating the data warehouse with record insertions, which avoids record locking and reduces the time required for data integration. Experimental evaluations using the TPC-H benchmark demonstrate the functionality of the proposed method. The results indicate that continuous data integration is feasible with an acceptable increase in query execution time, which is the trade-off for achieving real-time capabilities in the data warehouse. Future work involves developing an ETL tool to integrate the methodology with extraction and transformation routines for OLTP systems and further optimizing query instructions.

Effectively supporting real-time data integration in data warehouses is essential for real-time enterprises requiring up-to-date decision support. Traditional data warehouses have static structures and are not designed for continuous data integration, resulting in outdated data. To address this, the authors [1] propose a methodology that adapts data warehouse schemas and OLAP queries to enable continuous loading with minimal impact on query execution time. This is achieved using techniques such as table structure replication and query predicate restrictions. The efficiency of this method is demonstrated through the TPC-H benchmark, showing improved performance with lower transaction rates. The methodology focuses on adapting data warehouse schemas, ETL loading procedures, OLAP query adaptation, and data warehouse packing and reoptimization. It ensures minimal data update time windows, thereby maximizing data availability and minimizing negative performance impacts. However, the methodology may not be applicable in contexts where defining additive attributes for fact tables is challenging. The study presents a methodology for implementing Real-Time Data Warehousing (RTDW) by enabling continuous data integration with minimal query execution impact. The methodology was tested using the TPC-H benchmark across various scenarios with different transaction rates and data warehouse sizes. Results indicate that system performance is dependent on transaction rates and available RAM, with an average increase in query response time as the trade-off for real-time capability. The methodology proved to be scalable, demonstrating that real-time data warehousing is achievable. Future work includes developing an ETL tool to integrate the methodology with OLTP systems and optimizing query instructions.

SCANRAW, a novel database physical operator designed for in-situ processing over raw files, seamlessly integrates data loading and external tables while maintaining optimal performance across query workloads and ensuring zero time-to-query [9]. Leveraging a parallel super-scalar pipeline architecture, SCANRAW overlaps execution stages, including speculative loading, which utilizes additional I/O bandwidth during conversion to expedite subsequent queries. Dynamic adjustment to available system resources allows SCANRAW to efficiently utilize CPU cycles and I/O bandwidth. Implemented within a state-of-the-art database system, SCANRAW has been evaluated using both synthetic and real-world datasets, demonstrating optimal performance for query sequences and maximizing resource utilization without

disrupting normal query processing. The speculative loading feature ensures optimal performance across a sequence of queries, performing as efficiently as external tables for the initial query and surpassing database processing efficiency in the long run. This feature also enhances the efficiency of full data loading, enabling database processing with pre-loading to outperform external tables even for a two-query sequence. The architecture effectively handles CPU-intensive tasks, making these executions I/O-bound and ensuring optimal resource utilization. By parallelizing the conversion from text to binary, SCANRAW outperforms other data processing tools like BAMTools, leading to significant improvements in processing time. Overall, SCANRAW represents a significant advancement in in-situ data processing, offering a scalable and efficient solution for handling large raw data files. Future work will focus on extending support for multi-query processing over raw files.

Invisible Loading is a system designed to reduce the "time-to-first-analysis" in commercial analytical database systems by incrementally loading and organizing data from raw files into database systems while simultaneously processing the data using MapReduce jobs [5]. This system leverages MapReduce jobs' parsing and tuple extraction operations, allowing for immediate data analysis with minimal upfront effort and progressively enhancing the long-term performance benefits of database systems. By piggybacking on MapReduce jobs, Invisible Loading separates parsing code from data-processing code, reducing loading overhead by only copying vertical and horizontal partitions of data. It also introduces an Incremental Merge Sort technique for data reorganization, managing different columns at various stages of loading or reorganization. The system requires minimal human intervention and does not noticeably increase response time due to loading costs, making it a flexible and efficient solution for processing large, structured data sets typically stored in flat files on a file system. Additionally, the paper explores incremental data reorganization strategies, such as incremental merge sort and database cracking, and their integration with lightweight compression to improve query performance, especially over low cardinality data sets. Experimental results demonstrate that Invisible Loading offers cumulative performance comparable to pre-loading all data in advance, with minimal impact on MapReduce jobs.

In [14], a comprehensive guide is presented on loading data into Hive, which functions as a SQL layer on Hadoop. The guide encompasses essential tools and considerations necessary for this process. Key topics covered include designing the layout of the HDFS filesystem, selecting optimal schema and data formats to enhance performance, and choosing suitable compression algorithms. Different loading patterns and tools are examined, including Ambari Files View, Hadoop Command Line, Sqoop, and Apache Nifi, each offering unique use cases and advantages. The paper details how to make data accessible in Hive through external tables or the LOAD DATA statement. The process of loading incremental changes into Hive through delta files and new partitions is also outlined, emphasizing the ability to update data without modifying existing partitions. The Hive streaming API is introduced, enabling continuous ingestion of data into Hive tables or partitions, and ensuring immediate visibility to subsequent queries. However, limitations are noted, such as the necessity for the target table to be bucketed, stored in ORC format, and the need to enable ACID functionality with specific parameters. The paper concludes

by discussing the ongoing need for advancements in real-time data ingestion and accessibility, highlighting efforts by RDBMS vendors to develop plug-ins for their CDC technologies.

In [6], Hive, a MapReduce-based data warehouse, is evaluated for managing scientific data using the SSDB benchmark. The study illustrates that Hive can achieve acceptable performance for specific data analysis tasks compared to high-efficiency parallel databases, although it necessitates adjustments in storage configurations and indexing mechanisms. The authors detail the methodology for migrating SSDB to Hive, including query implementation and performance tuning strategies. The evaluation provides insights into the advantages and limitations of Hadoop/Hive for scientific data management, proposing avenues for further research and development. Additionally, the paper examines the impact of key factors such as parallel processing slot allocation, HDFS block size, data partitioning strategies, and row group size on performance outcomes. This analysis contributes to understanding the applicability of MapReduce-based frameworks in scientific data processing applications. The paper contextualizes these findings within the evolution of data processing systems, focusing on MapReduce-based platforms like Hadoop and their relevance in scientific data management compared to traditional database systems. It discusses benchmarking approaches specific to scientific data processing. Ultimately, the paper concludes that with targeted optimizations, MapReduce-based systems can achieve satisfactory performance levels for scientific data analysis tasks. It also highlights SciDB, a distributed database tailored for scientific array data, as a promising alternative with notable performance capabilities.

The impact of data organization and modeling strategies on processing times within Big Data Warehouses (BDWs) implemented using Hive is investigated in [15]. The study employs the SSB benchmark to compare multidimensional star schemas and fully denormalized tables across different Scale Factors (SFs), analyzing the influence of effective data partitioning. The findings reveal that fully denormalized structures generally outperform multidimensional approaches in terms of query execution times, particularly as data volumes scale up. Data partitioning significantly enhances query performance for both star schema and denormalized table configurations. Overall, the study concludes that while implementing multidimensional Data Warehouses (DWs) in Hive is feasible, fully denormalized data models offer superior performance benefits in Hadoop-based BDWs. It emphasizes the importance of strategic data partitioning based on frequently queried attributes to optimize query execution times effectively. The paper underscores the need for future research to explore advanced data partitioning and bucketing strategies, as well as guidelines for materialized views in the context of Big Data Warehousing. Additionally, it acknowledges the trade-offs between storage space and query performance inherent in denormalized tables, affirming that the performance gains typically outweigh the associated storage costs.

The performance analysis of Meteorological and Oceanographic (MOData) data on the Hive big data platform is presented in [13]. The study emphasizes the importance of proper data formatting, loading procedures, and analytical techniques to achieve efficient data analytics using Hive, compared to traditional database systems. Cus-

tom Serialization and Deserialization (SerDe) methods were developed specifically for MOData to align with Hive's data format requirements. Data loading and optimization were facilitated using a bash script. The study evaluated the performance of three query types across indexed and non-indexed tables, spanning data sizes from 20GB to 1TB. The results indicate that Hive offers superior response times compared to traditional databases, particularly for Type 1 queries where indexed tables consistently outperformed non-indexed ones across varying data sizes. The findings suggest potential benefits for industries such as oil and gas through the adoption of big data technologies like Apache Hadoop and Hive for exploratory data analysis. Future research directions could include optimizing query response times by minimizing the number of Mappers and exploring the application of meta-heuristic algorithms tailored to MOData within the Hadoop ecosystem. These efforts aim to further enhance the efficiency and scalability of data processing for complex scientific datasets.

The study presented in [4] explores the utilization of Hive and Hadoop for data processing within Facebook's infrastructure. It provides a comprehensive overview of the system's architecture, optimization techniques, and execution engine. Special attention is given to the challenges associated with resource scheduling, advocating for the use of separate clusters to handle ad-hoc queries versus reporting queries efficiently. Furthermore, the paper delves into Hive's integration with traditional warehousing tools and outlines ongoing improvements. It discusses future research directions, including the development of performance benchmarks and enhancements aimed at improving compatibility with commercial Business Intelligence (BI) tools.

The paper [8] addresses the complexities associated with Big Data integration, particularly focusing on the diverse range of data formats and the imperative for technologies that facilitate accessible and actionable data for decision-making. It proposes a semantic Extract-Transform-Load (ETL) framework leveraging semantic technologies to harmonize data from multiple sources into open linked data. Central to this framework is the establishment of a semantic data model utilizing RDF as the graph data model and SPARQL as the query language. Furthermore, the paper reviews existing literature on ETL processes and semantic ETL, and it presents a prototype implementation aimed at integrating public datasets on household travel and fuel economy. The objective is to enable meaningful data integration to support innovative Big Data applications and analytics. The discussions provided encompass various facets of Big Data, including techniques for graph analytics, data provenance, and the utilization of interlinked RDF data stores for learning classifiers. Additionally, the paper mentions several tools and platforms pivotal for data integration, such as IBM InfoSphere, Oracle Warehouse Builder, Microsoft SQL Server Integration Services, Informatica, Talend Open Studio, Pentaho, Protégé Ontology Editor, Oxygen XML Editor, and Apache Jena.

[19] conducts a systematic literature review focusing on Big Data Extraction, Transformation, and Loading (ETL) processes. It explores implementation methodologies, quality attributes, challenges, and the extent of coverage in current research. Key findings include:

1. Current ETL implementation techniques predominantly emphasize conceptual modeling, with limited incorporation of emerging technologies such as artificial intelligence and machine learning.

2. Data complexity and heterogeneity pose significant challenges in ETL, exacerbated by the ongoing growth in data volume and diversity.

3. Critical quality attributes like fault-tolerance and reliability are not sufficiently addressed in both ETL research and practical applications.

4. There is a recognized gap in research concerning emerging trends and implementation strategies in ETL, particularly those leveraging new technologies.

5. The application of ETL approaches across specific industries and geographic regions is underrepresented, indicating a need for broader exploration in research and practice.

6. ETL-related research and publications have shown a decline, possibly due to the emergence of alternative computing technologies overshadowing traditional ETL methods.

7. Future research should prioritize developing solutions to these challenges, including the integration of innovative technologies into ETL processes and enhancing data security throughout the ETL lifecycle.

dbX is a parallel SQL database designed for high performance on commodity hardware and cloud systems. It utilizes a vector execution model that leverages parallelism across multiple levels, including IO, data partitioning, intra-operator, operator, and intra-query [12]. Several techniques are proposed for optimizing load performance, such as parallel IO, file system & big IO, parallel task scheduling, minimal locking, transaction logging, data distribution, parallel loads, and the handling of constraints and indexes. Issues affecting data load and extract performance are examined, including synchronous IO, kernel buffering, IO contention, disk fragmentation, compute load, locking, ACID compliance, integrity constraints, data partitioning, parallel configuration, and error handling. Performance results from dbX on Amazon cloud and commodity systems demonstrate linear scale-up with cluster scale-out. The study also explores the impact of concurrent clients on load rates and the sustainability of load rates over extended periods and with larger file sizes. The need for further research and development in optimizing database load and extraction for the big data era is emphasized, highlighting the ongoing challenges and opportunities in this field.

A comprehensive review of popular Big Data processing tools—Drill, HAWQ, Hive, Impala, Presto, and Spark—emphasizes the importance of efficiently managing massive and complex data volumes [17]. Evaluating these tools using the TPC-H benchmark, the review discusses their architectures, functionalities, and performance

across different workloads and query types. SQL-on-Hadoop systems are highlighted for their user-friendly SQL interface, stressing the need to choose the right tool for specific analytical requirements. The tools are compared based on scalability, throughput, parallelism, SQL support, distributed architecture, fault tolerance, and machine learning capabilities. Experimental results show HAWQ and Presto as the fastest tools for a 10 GB dataset, outperforming others significantly. Impala performs well but struggles with complex queries, while Hive generally surpasses Spark except in complex joins. The review also examines the influence of file formats like ORC and Parquet on performance, noting their efficient compression and encoding. No single tool fits all Big Data processing needs; each has strengths and weaknesses based on the use case. The review covers various aspects of Big Data tools within the Hadoop ecosystem, including their application in diverse contexts such as stock market analysis and scientific image analytics. The evolution of Hadoop-based systems is explored, focusing on SQL database integration and query engines. The review underscores the importance of selecting appropriate tools for Big Data analytics, considering performance, scalability, and ease of use.

Handling and analyzing large volumes of data in the era of the Internet of Things (IoT) presents significant challenges. To address these, a user-friendly system called BigLoader is proposed [11]. BigLoader is designed to assist users in loading data from various sources into the most suitable NoSQL system. BigLoader enables users to specify the conceptual schema of the data to be loaded and identify the sources from which the data should be gathered. It also outlines intelligent strategies for selecting the most appropriate NoSQL system for deploying the conceptual schema. The emphasis is on creating a system that centers on the user, integrating data management strategies, smart and social recommendation services, innovative user interaction approaches, and intelligent systems for choosing the most suitable NoSQL system. A nested data model design is discussed, which is more intuitive for users than normalization, can handle inherently nested data, and is compatible with various formats. The need for ongoing research and development in optimizing database load and extraction for the big data era is highlighted, stressing the importance of continued innovation in this field.

Chapter 3

Background Study

3.1 Apache Hive

Apache Hive is a data warehousing and SQL-like query language system built on top of Apache Hadoop. It facilitates the reading, writing, and management of large datasets residing in distributed storage using SQL-like syntax. Hive is designed for batch processing and is widely used for data summarization, querying, and analysis of big data. It has revolutionized how data is processed and analyzed in the big data ecosystem. Its SQL-like interface, scalability, and integration with the Hadoop ecosystem make it an essential tool for handling large datasets. Despite its limitations in real-time processing, its strengths in batch processing, data warehousing, and ETL make it a popular choice for businesses handling vast amounts of data.

3.1.1 Key Features of Apache Hive

1. **SQL-Like Interface (HiveQL):** Hive provides an SQL-like query language called HiveQL (Hive Query Language), which is similar to SQL and makes it accessible to database professionals. It allows data analysts to perform queries without deep knowledge of Java or MapReduce.

2. **Schema on Read:** Unlike traditional databases that use a schema-on-write approach, Hive uses schema-on-read. This approach applies the schema when reading data, allowing flexibility in data storage formats and structures.

3. **Integration with Hadoop Ecosystem:** Hive integrates seamlessly with Hadoop, utilizing HDFS for storage and YARN for resource management. It can run on top of other Hadoop-compatible file systems, including Amazon S3, and supports execution engines like Apache Tez and Apache Spark.

4. **Scalability and Fault Tolerance:** Built on Hadoop, Hive inherits Hadoop's ability to scale horizontally across thousands of nodes and its fault-tolerant nature, which ensures data reliability during query execution.

5. **Extensibility through UDFs and Custom Functions:** Users can define their functions (UDFs) to perform specific data processing tasks, extending Hive's capabilities beyond its standard functions.

6. **Partitioning and Bucketing:** Hive supports data partitioning to manage large datasets by breaking them into smaller, manageable parts. Bucketing further segments data within partitions, leading to more efficient query execution.

3.1.2 Hive Architecture

3.1.2.1 Architecture Components

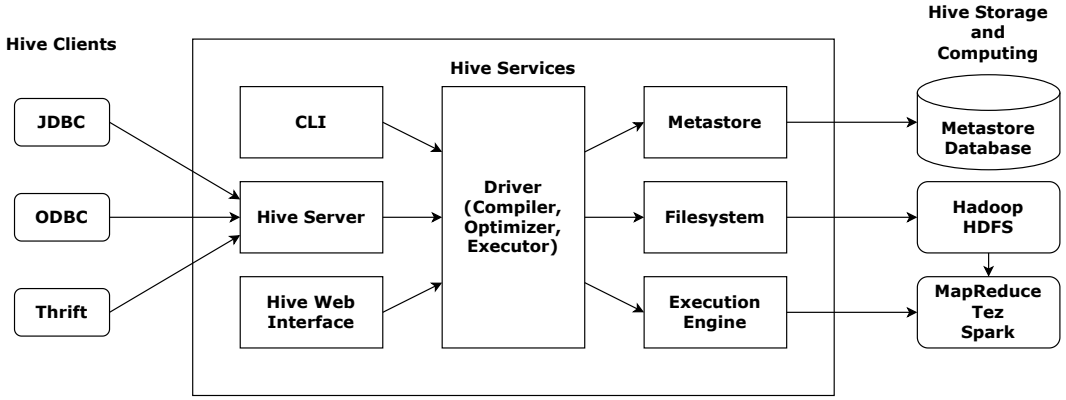

Figure 3.1: Hive Architecture

Figure 3.1 shows a detail architecture of Apache Hive [20].

- **User Interface (UI):** Hive provides a CLI, Web UI, and JDBC/ODBC interfaces for users to submit queries.
- **Driver:** The driver receives the queries from the user interface, parses them, and manages the lifecycle of a query.
- **Compiler:** Converts HiveQL statements into directed acyclic graphs (DAGs) of MapReduce, Tez, or Spark tasks.
- **Metastore:** A critical component of Hive, the Metastore stores metadata about tables, partitions, schemas, and other objects.
- **Execution Engine:** The engine manages the execution of tasks generated by the compiler using MapReduce, Tez, or Spark, coordinating with Hadoop to process data.
- **HDFS/YARN:** HDFS is the primary storage layer for Hive, while YARN manages resource allocation across the Hadoop cluster.

3.1.3 Hive Table Types

Apache Hive is an integral part of modern big data ecosystems, designed to manage and query large datasets stored in Hadoop Distributed File System (HDFS). Hive offers two primary types of tables for organizing data: **Managed** (also known as **Internal**) tables and **External** tables. Understanding the differences between these two types is crucial for designing an efficient data architecture, especially in the context of optimizing data loading workflows.

3.1.3.1 Managed Tables

In Hive, **Managed Tables** are tables where Hive assumes full responsibility for the data. When a managed table is created, Hive stores both the metadata (table schema) and the actual data in specific directories within the Hive warehouse directory (typically located at /user/hive/warehouse/).

- **Characteristics of Managed Tables:**
  - **Data Ownership:** Hive owns the data in the managed table. If the table is dropped, both the metadata and the actual data files in HDFS are deleted.
  - **Data Storage:** Data is stored in Hive's warehouse directory by default. Hive automatically manages the physical location of the data.
  - **Ease of Management:** Managed tables are easier to manage when the user wants Hive to control the lifecycle of the data. All data management tasks (like insertions, deletions, and modifications) are handled by Hive internally.
  - **ETL Usage:** Managed tables are commonly used in ETL (Extract, Transform, Load) processes where data is ingested into Hive, processed, and stored permanently for analysis.
- **Example of Managed Table Creation:**

```
CREATE TABLE sales_data (
    sale_id INT,
    product STRING,
    quantity INT,
    price DOUBLE
) STORED AS ORC;
```

  In the above example, Hive stores the data for the sales_data table in its default warehouse directory. Dropping the table will also remove the data from the warehouse.

- **Key Use Cases:**
  - **Permanent Data Storage:** When data is created, transformed, and intended to remain in Hive.
  - **Complete Data Lifecycle Control:** When Hive is responsible for both storing and managing the data lifecycle, including purging old data.
  - **Tight Data Coupling:** Suitable when Hive is the only system accessing the data, and the data does not need to be shared externally.

3.1.3.2 External Tables

External Tables, on the other hand, provide a more flexible way to manage data. These tables allow Hive to reference data stored at an external location in HDFS or even outside HDFS (e.g., Amazon S3). With external tables, Hive only manages the

schema (metadata), while the data itself is stored and managed outside of Hive's control.

- **Characteristics of External Tables:**
  - **Data Ownership:** Hive only manages the metadata, while the external data source (e.g., HDFS, S3) owns and manages the data itself. Dropping the table does not affect the underlying data; it simply removes the metadata.
  - **Data Storage:** The data is stored in a location specified by the user when the table is created. This location can be within HDFS, another Hadoop cluster, or even a cloud-based storage system.
  - **Data Sharing:** External tables are useful when data needs to be shared between Hive and other systems, such as custom MapReduce jobs or external analytics engines.
  - **Decoupling Data and Metadata:** External tables provide flexibility by decoupling the data from Hive's warehouse, allowing the data to be used by multiple systems.
- **Example of External Table Creation:**

```
CREATE EXTERNAL TABLE product_data (
    product_id INT,
    name STRING,
    category STRING,
    price DOUBLE
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/product_data/';
```

  In this example, the product_data table refers to data located at /data/product_data/. Dropping this table will not delete the data stored in HDFS at that location.

- **Key Use Cases:**
  - **Shared Data:** When the data needs to be accessed and shared by other systems (e.g., Spark, Flink, external applications).
  - **Existing Data Sources:** When the data already exists in HDFS or an external system and there is no need to move it into Hive's warehouse directory.
  - **Data Reusability:** External tables are useful when the same data is to be used across multiple environments (e.g., multiple Hive clusters or environments).

Aspect	Managed Table	External Table
Data Location	Data stored in Hive's warehouse directory.	Data stored in an external location, as specified by the user.
Data Ownership	Hive owns and manages the data.	Data is owned and managed externally (by the user or another system).
Table Drop Behavior	Deleting the table deletes both the data and the metadata.	Deleting the table only removes the metadata; the data remains intact.
Use Case	Permanent data storage controlled entirely by Hive.	Reference existing data stored outside of Hive, allowing sharing with other systems.
Data Deletion	Dropping the table removes the data.	Data is not deleted upon dropping the table.
Best for	Complete Hive data management.	Sharing data across different systems or reusing existing datasets.

Table 3.1: Comparison of Hive Managed and External Tables

3.1.3.3 Comparison of Managed and External Tables

The table 3.1 compares the key aspects of Managed and External tables in Hive:

3.1.3.4 Role of Managed and External Tables in ETL

In the context of ETL (Extract, Transform, Load) processes, both managed and external tables play crucial roles:

- **Managed Tables** are often used for storing **intermediate or final processed data** in Hive, where the data's lifecycle is tightly controlled, and no external systems need to access it.
- **External Tables** are typically used to **import data** from various external sources (e.g., raw logs, CSV files, data from cloud storage) into Hive for further transformation. They are also used when Hive is integrated with other data processing tools like Spark or Flink.

3.1.4 Hive Data Model

Hive organizes data into tables, which are similar to those in relational databases. Tables are further divided into partitions and buckets, which optimize data processing and querying. Relationships between hive tables, partitions and buckets are shown in Figure 3.2.

- **Tables:** Analogous to RDBMS tables, stored in HDFS.
- **Partitions:** Logical divisions of tables, improving query speed.

- **Buckets:** Further subdivision of data within partitions for better optimization.

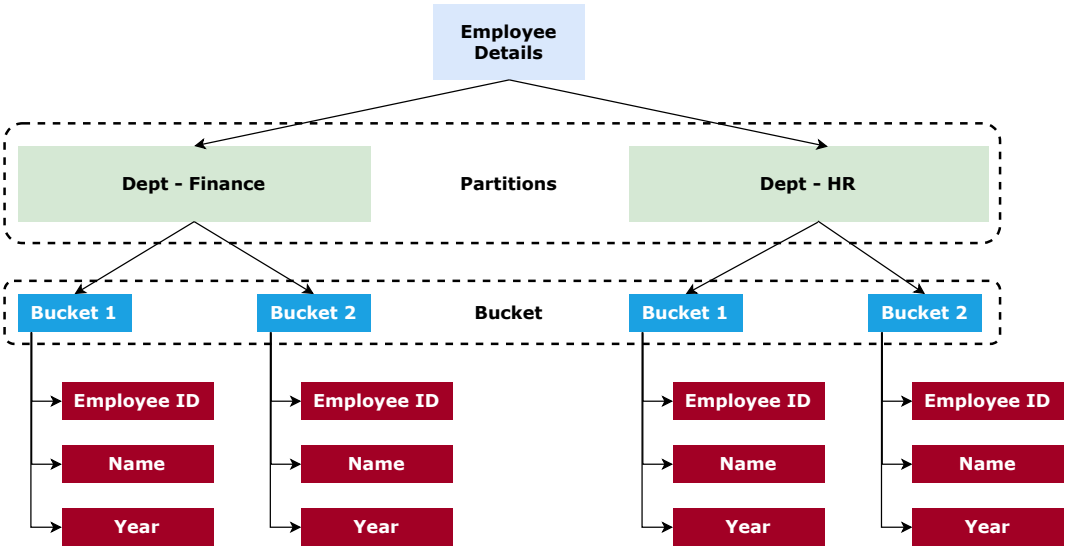

Figure 3.2: Hive Data Model

3.1.5 Common Use Cases

1. **Data Warehousing and Analytics:**
   Hive is often used to create large-scale data warehouses, allowing users to run queries on massive datasets for business analytics.

2. **ETL Processes:**
   Hive is a critical tool in ETL (Extract, Transform, Load) pipelines, processing raw data into structured forms for further analysis.

3. **Log Processing:**
   Hive is widely used to process logs generated by web applications, transforming raw logs into structured data for reporting.

4. **Business Intelligence and Reporting:**
   Hive's SQL-like interface allows integration with BI tools, making it suitable for data analysis and visualization.

3.1.6 Advantages and Limitations

3.1.6.1 Advantages

- **Scalable and Cost-Effective:** Hive scales easily, providing cost-effective processing of big data.
- **Compatibility with BI Tools:** Hive's SQL-like syntax allows easy integration with popular BI tools like Tableau and Power BI.
- **Schema Flexibility:** Supports various data formats including text, ORC, Parquet, Avro, and more.

3.1.6.2 Limitations

- **Not Suitable for Real-Time Processing:** Hive is designed for batch processing and is not ideal for real-time data processing needs.
- **Performance Overheads:** Queries in Hive can be slower compared to other SQL-based engines because of the overhead of MapReduce.

3.2 Apache Hive Data Loading Architecture

Apache Hive is a data warehousing solution built on top of Hadoop, providing SQL-like query capabilities for managing and analyzing large datasets. Loading data into Hive involves several steps and architectural components to ensure that the data is ingested, stored, and managed efficiently.

3.2.1 Data Sources

The data to be loaded into Hive can come from various sources, such as:

- Local file systems
- Distributed file systems (e.g., HDFS)
- Relational databases
- NoSQL databases
- Streaming data sources

3.2.2 Key Components of Hive Data Loading

1. **Hive Metastore:**
    - The Metastore is the central repository for all metadata concerning Hive tables, including table definitions, column types, partition details, and the location of data files.
    - It acts as a catalog that enables Hive to map SQL queries to data stored in Hadoop Distributed File System (HDFS) or other supported file systems.
2. **Hive Warehouse Directory:**
    - This is the default location where Hive stores its managed table data. By default, it is located at /user/hive/warehouse on HDFS.
    - For external tables, data can reside outside the warehouse directory, providing flexibility in managing the data independently.
3. **Hive Tables:**
    - **Managed Tables:** Hive manages both the schema and the data. Data files are moved into the Hive warehouse directory upon loading.
    - **External Tables:** Only the schema is managed by Hive, allowing the data to reside outside Hive's control, such as on HDFS or a cloud storage system.

3.2.3 Data Loading Methods in Hive

1. **LOAD DATA Command:**

   - The LOAD DATA command is used to move data files into Hive tables from a specified location, either on the local file system or HDFS.
   - Syntax:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE]
INTO TABLE table_name [PARTITION (partcol1=val1,
   partcol2=val2 ...)];
```

   - Example:

```
LOAD DATA LOCAL INPATH '/path/to/datafile.csv'
INTO TABLE employees;
```

   - **LOCAL** keyword specifies that the file is on the local file system; otherwise, Hive looks for the file on HDFS.
   - **OVERWRITE** will replace existing data in the table if specified.

2. **INSERT INTO/INSERT OVERWRITE:**

   - These SQL commands allow inserting data into existing Hive tables using results from queries.
   - **INSERT INTO** appends data to a table, while **INSERT OVERWRITE** replaces existing data.
   - Example:

```
INSERT INTO employees SELECT * FROM new_employees;
```

```
INSERT OVERWRITE TABLE employees
SELECT * FROM updated_employees
WHERE status = 'active';
```

3. **CREATE TABLE AS SELECT (CTAS):**

   - The CTAS statement creates a new table and directly inserts the results of a query into it.
   - This approach is often used to create derived tables or to transform and load data in a single step.
   - Example:

```
CREATE TABLE high_salary_employees
AS
SELECT * FROM employees
WHERE salary > 100000;
```

4. **Using Hive Streaming API:**

   - The Hive Streaming API allows continuous data ingestion into Hive tables, particularly useful for near real-time analytics.
   - It supports incremental data loading, where data is streamed into Hive tables without needing batch processing.

5. **Bulk Import with Apache Sqoop:**

   - Apache Sqoop is often used for bulk data import from relational databases into Hive.
   - Data is imported directly into Hive tables using Sqoop commands, facilitating efficient loading of large datasets.

3.2.4 Partitioning and Bucketing in Hive

1. **Partitioning:**

   - Partitioning divides a table into parts based on the values of a specific column, improving query performance by limiting the amount of data scanned.
   - For example, partitioning by date or region allows queries to focus only on relevant partitions.
   - Example:

```
CREATE TABLE sales (
    product STRING,
    amount INT, date STRING
)
PARTITIONED BY (region STRING);
```

2. **Bucketing:**

   - Bucketing further divides data within each partition into buckets based on the hash of a column, which helps improve query performance during joins and sampling.
   - Example:

```
CREATE TABLE orders (
    orderid INT,
    customer_name STRING
)
CLUSTERED BY (customerid)
INTO 4 BUCKETS;
```

3.2.5 File Formats and Compression

1. **File Formats:**
   Choosing the right file format is critical for performance optimization. Table 3.2 compares various file formats used in Hive, highlighting their properties such as columnar support, compressibility, readability, handling of complex data types, schema evolution, space utilization, and support for ACID transactions. Here's a detailed breakdown of each format and its capabilities:

File Format/ Properties	Columnar	Compressible	Readable	Complex Data Types	Schema Evolution	Space Utilization Rank	ACID Transactions
Text File	No	Yes	Yes	No	No	6	No
Sequence File	No	Yes	Yes	Yes	Yes	5	No
RC File	Yes	Yes	No	Yes	Yes	2	No
ORC File	Yes	Yes	No	Yes	Yes	1	Yes
Avro	No	Yes	No	Yes	Yes	4	No
Parquet	Yes	Yes	No	Yes	Yes	3	No

Table 3.2: Comparison of File Formats and Properties

   (a) **Text File:**
   - **Columnar:** Text files store data in rows, making them inefficient for column-based queries.
   - **Compressible:** Text files can be compressed using algorithms like Gzip or Snappy, though not as efficiently as columnar formats.
   - **Readable:** Plain text is human-readable, making it easy to debug or inspect the data.
   - **Complex Data Types:** Does not natively support complex data types like structs, arrays, or maps without additional serialization.
   - **Schema Evolution:** Schema changes are not natively supported; modifications need careful data handling and manual adjustments.
   - **Space Utilization Rank:** Least space-efficient among the formats, as plain text takes up more storage without any inherent data optimization.
   - **ACID Transactions:** Does not support ACID properties, making it unsuitable for transactional processing.

   (b) **Sequence File:**
   - **Columnar:** Data is stored in a binary format with key-value pairs but not in a columnar layout.
   - **Compressible:** Supports compression at the record and block levels, which can improve performance by reducing I/O overhead.
   - **Readable:** Binary format is readable through Hadoop tools but not as straightforward as plain text.
   - **Complex Data Types:** Can handle complex data types, making it versatile for various data structures.
   - **Schema Evolution:** Supports changes to the data schema without needing to rewrite existing data.

- **Space Utilization Rank:** More efficient than plain text but less optimized compared to columnar formats.
- **ACID Transactions:** Does not support ACID properties, limiting its use in transactional systems.

(c) **RC File (Record Columnar File):**

- **Columnar:** RCFile stores data in a columnar format, making it efficient for queries that access specific columns.
- **Compressible:** Supports compression, which helps in reducing the storage footprint and speeding up query performance.
- **Readable:** Not easily readable due to its binary format and columnar storage structure.
- **Complex Data Types:** Capable of storing complex data types such as structs, arrays, and maps.
- **Schema Evolution:** Allows schema modifications, providing flexibility in data handling over time.
- **Space Utilization Rank:** Highly efficient in space utilization due to its columnar nature and compression.
- **ACID Transactions:** Lacks native support for ACID properties, making it less suitable for applications requiring strong consistency.

(d) **ORC File (Optimized Row Columnar File):**

- **Columnar:** ORC is a columnar storage format, optimized for fast data retrieval, especially in analytic workloads.
- **Compressible:** ORC files provide efficient compression, reducing both storage costs and read times.
- **Readable:** Not human-readable; requires Hive or other tools for data interpretation.
- **Complex Data Types:** Fully supports complex data types, making it ideal for structured data analysis.
- **Schema Evolution:** Supports adding or changing columns without needing to rewrite the entire dataset.
- **Space Utilization Rank:** The most space-efficient format due to advanced compression techniques and columnar storage.
- **ACID Transactions:** ORC supports ACID properties, making it suitable for applications requiring transactional integrity.

(e) **Avro:**

- **Columnar:** Avro stores data in a row-oriented format, focusing on efficient data serialization.
- **Compressible:** Supports various compression algorithms, which help minimize the storage and improve data transfer speeds.
- **Readable:** Data is stored in a binary format, which is not human-readable but optimized for machine processing.
- **Complex Data Types:** Strong support for complex data structures, making it suitable for varied data types.

- **Schema Evolution:** Avro excels in handling schema evolution, allowing forward and backward compatibility for schema changes.
- **Space Utilization Rank:** Efficient but not as optimized as columnar formats like ORC or Parquet.
- **ACID Transactions:** Does not support ACID transactions, limiting its use in environments requiring strong consistency guarantees.

(f) **Parquet:**

- **Columnar:** Parquet is a highly efficient columnar storage format that optimizes both read and write performance.
- **Compressible:** Supports compression, which helps to significantly reduce the storage footprint and improve I/O performance.
- **Readable:** Data is stored in a columnar binary format that requires specific tools for access.
- **Complex Data Types:** Capable of storing complex data types, enhancing its use in data warehousing and analytics.
- **Schema Evolution:** Allows schema changes, making it versatile for evolving data models.
- **Space Utilization Rank:** Highly efficient in space utilization, especially for large, complex datasets.
- **ACID Transactions:** Does not support ACID properties, which restricts its use in transactional applications.

Key Insights:

- **Columnar Formats (ORC, Parquet, RCFile):** Best suited for analytical workloads due to their ability to store data in columns, allowing efficient query processing and reduced I/O.
- **Compressibility:** All formats support compression, but columnar formats typically offer better compression rates due to their data organization.
- **Schema Evolution:** Avro, ORC, and Parquet handle schema changes gracefully, which is crucial for data systems that undergo frequent updates.
- **Space Utilization:** ORC is the most space-efficient, followed by Parquet and RCFile, due to their advanced data compression and storage optimization techniques.
- **ACID Support:** Only ORC supports ACID transactions, making it the preferred choice for use cases that require transactional integrity.

This analysis helps to choose the appropriate file format in Hive based on the specific needs of storage efficiency, query performance, and schema management capabilities.

2. **Compression:**

   - Compression reduces storage space and can improve I/O performance by minimizing data transfer times.
   - Supported compression algorithms include Snappy, Gzip, and Bzip2, with Snappy being commonly used due to its balance between compression speed and size.

3.2.6 Schema Evolution and Data Management

1. **Schema Evolution:**

   - Hive supports schema evolution, allowing changes to table schemas without rewriting existing data. For example, adding new columns or changing data types is possible through the ALTER TABLE command.

2. **Table Management Commands:**

   - **ALTER TABLE:** Used for schema changes.
   - **MSCK REPAIR TABLE:** Updates the metadata for tables with newly added partitions.

3.2.7 Best Practices for Data Loading in Hive

1. **Use External Tables for Large Datasets:** This approach keeps data management independent of Hive, providing more control over data storage and deletion.

2. **Partition and Bucket Large Tables:** This strategy optimizes query performance by reducing the data read during queries.

3. **Choose Columnar File Formats for Analytics:** Formats like ORC and Parquet significantly improve query performance by reading only the necessary columns.

4. **Utilize Compression for Storage Efficiency:** Apply compression to reduce storage costs and improve data retrieval speeds.

5. **Maintain Metadata Consistency:** Regularly update the Hive Metastore, especially when working with external data sources, to ensure data accuracy.

3.2.8 Data Loading Workflow in Hive

1. **Extract Data:** Data is extracted from various sources such as databases, APIs, or flat files.

2. **Transform Data:** Data may be transformed using tools like Apache Spark, Pig, or ETL scripts before loading into Hive.

3. **Load Data:** The data is loaded into Hive using LOAD DATA, INSERT INTO, INSERT OVERWRITE, or other methods based on requirements.

4. **Optimize Storage:** Partitioning, bucketing, and file format optimization are applied to improve performance.

5. **Manage Data:** Regular schema updates and data maintenance are performed to ensure data consistency and accessibility.

3.3 Traditional Hive Data Ingestion Architecture (Any File Type)

This architecture outlines the traditional approach used in Hive to ingest data, typically stored in HDFS (Hadoop Distributed File System), and organize it into partitions for efficient querying. Here's a detailed step-by-step explanation of the architecture depicted in the Figure 3.3:

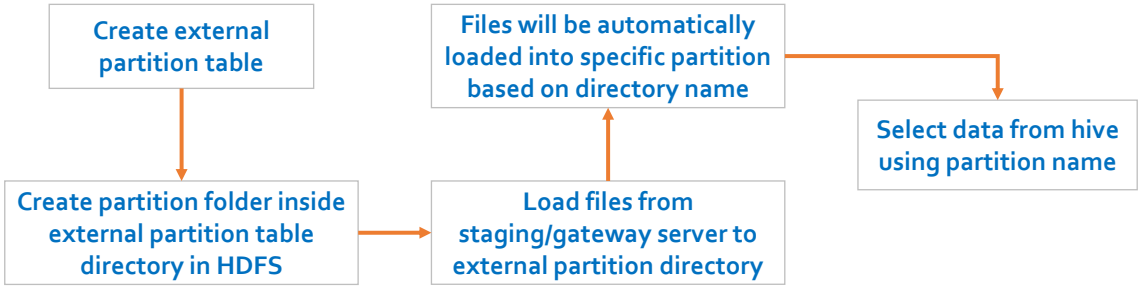

Figure 3.3: Traditional Hive Data Ingestion Architecture (Any File Type)

1. **Create External Partition Table:**

   - **Purpose:** An external table in Hive is defined, which points to data stored outside the Hive database in HDFS. This is useful because it allows data to be managed independently of Hive and does not delete data when the table is dropped.
   - **Benefits:** It allows you to work with large datasets stored in HDFS while maintaining flexibility in managing the table schema.

2. **Create Partition Folder Inside External Partition Table Directory in HDFS:**

   - **Purpose:** After creating the external table, you create directories (folders) in HDFS corresponding to the partitions of the table. Each partition represents a subset of the data, typically organized by a column (e.g., date, region) for efficient query performance.
   - **Example:** If the partition column is a date, folders like year=2023/month=09/day=04 would be created.

3. **Load Files from Staging/Gateway Server to External Partition Directory:**

- **Purpose:** Data files are loaded into the partition folders from a staging or gateway server. This staging server acts as an intermediary where data is initially uploaded or processed before being transferred to the HDFS partition directories.
- **Data Types:** The files can be of any type (e.g., CSV, JSON, Parquet), but the type must align with the table schema defined in Hive.

4. **Files Will Be Automatically Loaded into Specific Partitions Based on Directory Name:**
   - **Automation:** Hive automatically maps the files to the corresponding partitions based on the folder names in HDFS. This is a key feature of partitioning, as it enables Hive to only scan relevant data during queries.
   - **Benefit:** This reduces the amount of data read and processed, improving performance for large datasets.
5. **Select Data from Hive Using Partition Name:**
   - **Query Execution:** Once the data is loaded into the partitions, it can be queried using HiveQL (Hive Query Language). Queries can filter data based on partition columns, significantly reducing the query time.
   - **Example:** A query like

```
SELECT * FROM table_name
WHERE partition_column = '2023-09-04';
```

     will only scan the data in the relevant partition.

Key Advantages:

- **Improved Performance:** By partitioning data, Hive can significantly reduce the amount of data read during queries.
- **Scalability:** This architecture is highly scalable, allowing for the ingestion of large volumes of data.
- **Flexibility:** The use of external tables keeps the data management flexible, as the underlying data in HDFS can be independently managed.

Use Case:
This architecture is commonly used in data warehousing environments where large datasets are ingested periodically (e.g., daily logs) and need to be efficiently queried based on specific criteria like date or category.

3.4 Choosing Right File Type

For large-scale data warehouses or data lakes, the approach to data storage can significantly impact the performance, storage efficiency, and overall manageability of the system. Here's an explanation of why simply storing any type of file is not ideal, and why ORC (Optimized Row Columnar) is considered the best for transactional data, performance, and size.

3.4.1 Challenges of Storing Any Type of File

- **Lack of Optimization:** Different file formats (e.g., plain text, CSV) do not offer the necessary optimizations for reading, writing, and querying data efficiently. This can lead to slower performance when processing large volumes of data.
- **Inefficient Space Utilization:** Generic file formats are often not space-efficient. For instance, text files or CSVs store data in raw format without any compression, leading to higher storage costs, especially when dealing with terabytes or petabytes of data.
- **Complex Data Handling:** Many file types lack support for complex data types, such as nested structures, which are often required in modern data analytics.
- **Limited Support for Schema Evolution:** Not all file formats support schema evolution, making it difficult to handle changes in the data schema over time without impacting existing data.

3.4.2 Why ORC is Best for Transactional Data, Performance, and Size

- **Columnar Storage:** ORC is a columnar file format, which means that data is stored column by column rather than row by row. This allows for efficient data retrieval and is particularly advantageous for analytical queries that only need access to a few columns.
- **High Compression:** ORC offers high compression rates, which significantly reduce storage requirements. It uses advanced compression techniques that are optimized for columnar data, resulting in smaller file sizes compared to other formats like Parquet or Avro.
- **Improved Performance:** ORC is designed for high-performance read and write operations, which is crucial for transactional data. It uses indexes, lightweight compression, and in-file data statistics to speed up data retrieval, making it faster than other file formats for most analytical workloads.
- **Support for ACID Transactions:** ORC supports ACID (Atomicity, Consistency, Isolation, Durability) transactions, which is essential for ensuring data integrity and consistency in data lakes and warehouses where transactional data is frequently updated.

- **Schema Evolution:** ORC supports schema evolution, allowing changes to be made to the data schema without breaking existing queries or applications. This flexibility is critical in environments where data structures change over time.
- **Space Efficiency:** With its columnar layout, ORC achieves excellent space utilization, ranking as one of the most space-efficient formats, which helps reduce storage costs significantly in large-scale environments.

While it may seem convenient to store data in any format, using optimized formats like ORC is crucial in large-scale data warehouses or lakes. ORC's columnar storage, high compression, performance optimization, support for complex data types, and ACID compliance make it an ideal choice for managing large volumes of transactional data efficiently.

3.5 Traditional Hive Data Ingestion Architecture (Any File type to ORC)

In the realm of big data, efficient data ingestion and storage are paramount for ensuring optimal performance and scalability. Apache Hive, a data warehousing solution built on top of Hadoop, provides robust mechanisms for managing and querying large datasets. The Architecture given in Figure 3.4 delves into the traditional Hive data ingestion architecture, specifically focusing on the process of converting various file types into the ORC (Optimized Row Columnar) format. By leveraging this architecture, organizations can achieve significant improvements in data processing efficiency, storage optimization, and query performance. The following sections provide a detailed analysis of each step involved in this architecture, highlighting the key components and their roles in the data ingestion pipeline.

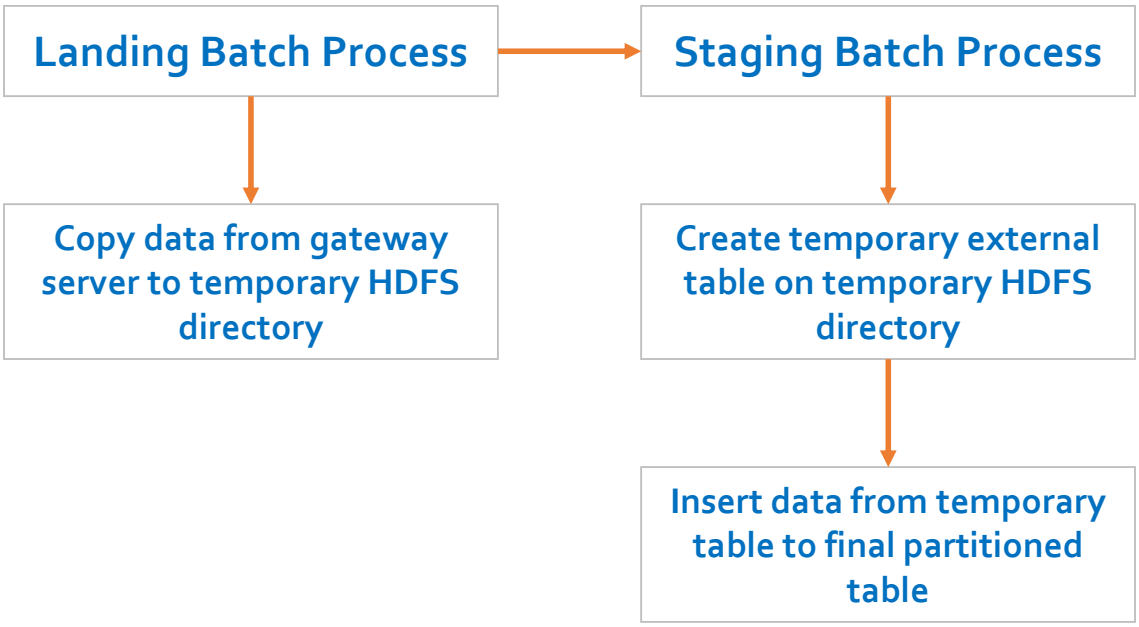

Figure 3.4: Traditional Hive Data Ingestion Architecture (Any File type to ORC)

3.5.1 Landing Batch Process

1. **Copy Data from Gateway Server to Temporary HDFS Directory**

   - **Description:** This initial step involves transferring raw data files from an external gateway server to a temporary directory within the Hadoop Distributed File System (HDFS).
   - **Purpose:**
     - **Data Ingestion:** This step ensures that the data is ingested into the Hadoop ecosystem, making it accessible for further processing.
     - **Temporary Storage:** The temporary HDFS directory acts as a staging area where data can be temporarily stored before it is processed and moved to its final destination.
   - **Process:**
     - Data is copied using tools like scp, rsync, or Hadoop-specific tools like DistCp (Distributed Copy).
     - The temporary directory is typically named to reflect the batch or date of ingestion for easy tracking.

3.5.2 Staging Batch Process

1. **Create Temporary External Table on Temporary HDFS Directory**

   - **Description:** An external table is created in Hive that points to the data stored in the temporary HDFS directory.
   - **Purpose:**
     - **Data Management:** This step allows Hive to manage and query the data without physically moving it from the temporary directory.
     - **Schema Definition:** The external table defines the schema of the data, specifying the structure and data types of the columns.
   - **Process:**
     - A HiveQL command is used to create the external table. For example:

```
CREATE EXTERNAL TABLE temp_table (
    column1 STRING,
    column2 INT,
    ...
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION 'hdfs://path/to/temporary/directory';
```

     - This command specifies the schema and the location of the data in HDFS.

2. **Insert Data from Temporary Table to Final Partitioned Table**

   - **Description:** The data from the temporary external table is inserted into a final partitioned table in Hive.
   - **Purpose:**
     - **Data Transformation:** This step involves transforming the data as needed (e.g., filtering, aggregating) before loading it into the final table.
     - **Optimized Storage:** The final table is partitioned and stored in the ORC format, which is optimized for performance and storage efficiency.
   - **Process:**
     - A HiveQL command is used to insert the data. For example:

```
INSERT INTO final_table PARTITION (
   partition_column)
SELECT column1, column2, ...
FROM temp_table;
```

     - This command transfers the data from the temporary table to the final partitioned table, applying any necessary transformations.
     - The final table is defined to use the ORC format for storage:

```
CREATE TABLE final_table (
    column1 STRING,
    column2 INT,
    ...
)
PARTITIONED BY (partition_column STRING)
STORED AS ORC;
```

3.5.3 Benefits of This Architecture

- **Scalability:** The architecture is designed to handle large volumes of data efficiently.
- **Performance:** Using ORC format for the final table ensures fast query performance due to its columnar storage and compression capabilities.
- **Flexibility:** The use of external tables allows for flexible data management and easy integration with various data sources.
- **Data Organization:** Partitioning the final table helps in organizing the data and improving query performance by reducing the amount of data scanned.

This architecture ensures that data is ingested, managed, and stored in an optimized manner, making it suitable for large-scale data warehouses or data lakes.

3.6 Sequential Process Bottlenecks

The sequential nature of the landing and staging batch processes can indeed slow down the ingestion of large volumes of data. Let's delve into why this happens and explore potential improvements.

1. **Landing Batch Process:**
    - **Copying Data:** This step involves transferring data from the gateway server to a temporary HDFS directory. The speed of this process depends on network bandwidth and the size of the data. Large datasets can take a significant amount of time to transfer.
2. **Staging Batch Process:**
    - **Creating Temporary External Table:** Once the data is in the temporary HDFS directory, an external table is created. This step is relatively quick but still adds to the overall processing time.
    - **Inserting Data into ORC:** This is the most time-consuming step. Converting data to ORC format involves reading the data from the temporary table and writing it into the final partitioned table. This process is I/O intensive and can be slow for large datasets.

3.6.1 Why Sequential Processing is Slow

- **Dependency:** Each step depends on the completion of the previous one. The landing process must finish before the staging process can begin. This sequential dependency creates a bottleneck, especially when dealing with large volumes of data.
- **Resource Utilization:** Sequential processing can lead to underutilization of resources. While one process is running, other resources may remain idle, leading to inefficiencies.

3.6.2 Potential Improvements

1. **Parallel Processing:**
    - **Data Partitioning:** Split the data into smaller chunks and process them in parallel. This can significantly reduce the time required for data transfer and conversion.
    - **Concurrent Execution:** Run the landing and staging processes concurrently where possible. For example, start processing the first chunk of data in the staging process while the next chunk is being copied in the landing process.
2. **Incremental Data Ingestion:**
    - Instead of waiting for the entire dataset to be copied, start processing data incrementally. This approach can help in reducing the overall processing time.

3. **Optimized Data Transfer:**

    - Use faster data transfer protocols and optimize network bandwidth to speed up the data copying process.

4. **Efficient Resource Management:**

    - Allocate resources dynamically based on the workload. Use resource management tools to ensure that the system is not overloaded and resources are utilized efficiently.

By implementing these improvements, one can significantly reduce the time required for data ingestion and make the process more efficient.

Chapter 4

Research Methodology

4.1 Proposed Architecture

4.1.1 Two Asynchronous Processes

A Proposed Architecture that separates the Landing Batch Process and Staging Batch Process into two asynchronous processes is shown in Figure 4.1. Let's break down how this new architecture works and its potential benefits:

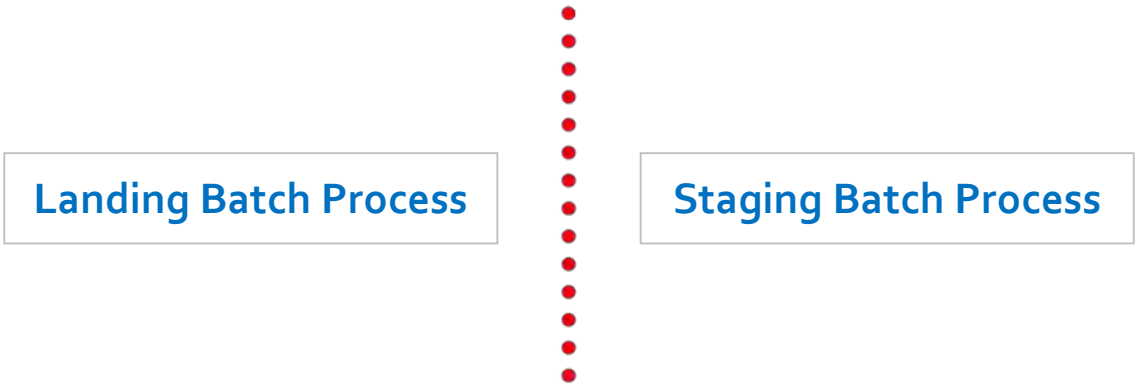

Figure 4.1: Proposed Architecture – Two Asynchronous Processes

1. **Landing Batch Process**
   - **Purpose:** This process is responsible for the initial data ingestion.
   - **Steps:**
     - **Copy data from gateway server to temporary HDFS directory:** Similar to the traditional approach, data is transferred from the gateway server to a temporary HDFS directory. However, this process runs independently of the staging process.
2. **Staging Batch Process**
   - **Purpose:** This process handles the transformation and loading of data into the final format.

- **Steps:**
  - **Create temporary external table on temporary HDFS directory:** Once the data is available in the temporary HDFS directory, an external table is created.
  - **Insert data from temporary table into Final Table:** Data is then inserted from the temporary table into the final partitioned table in ORC format.

4.1.1.1 Benefits of Asynchronous Processing

1. **Parallel Execution:**
   - By separating the landing and staging processes, they can run in parallel rather than sequentially. This reduces the overall processing time as both processes can utilize system resources simultaneously.
2. **Improved Resource Utilization:**
   - Resources are better utilized since both processes can run concurrently. This reduces idle time and ensures that the system is working efficiently.
3. **Scalability:**
   - The architecture can handle larger volumes of data more effectively. As data ingestion and processing are decoupled, each process can be scaled independently based on the workload.
4. **Flexibility:**
   - This approach allows for more flexibility in managing data workflows. For example, if there is a delay in data transfer, the staging process can still proceed with the available data without waiting for the entire dataset to be copied.
5. **Reduced Bottlenecks:**
   - By decoupling the processes, bottlenecks caused by sequential dependencies are minimized. This leads to faster data processing and ingestion times.

4.1.1.2 Flow of Processes

- **Landing Batch Process:** Runs independently to copy data from the gateway server to the temporary HDFS directory.
- **Staging Batch Process:** Runs independently to create a temporary external table and insert data into the ORC format.

This proposed architecture can significantly improve the efficiency and speed of data ingestion and processing, especially for large datasets.

4.1.2 Master Landing & Staging Batch Process

The proposed architecture outlines a Master Landing & Staging Batch Process that separates the responsibilities between a master process and multiple child processes is shown in Figure 4.2. Here's a detailed explanation:

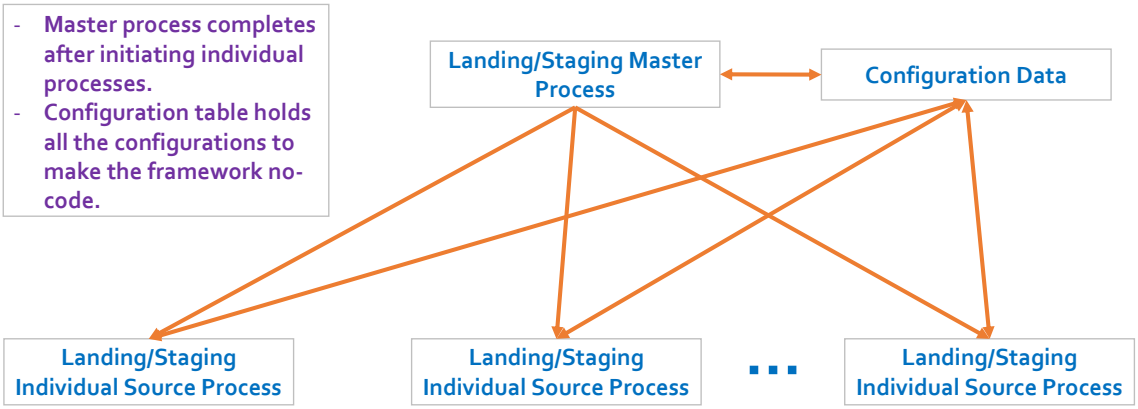

Figure 4.2: Master Landing & Staging Batch Process

1. **Landing/Staging Master Process**
   - **Role:** The master process is responsible for initiating and coordinating the data loading tasks for each individual source.
   - **Steps:**
     - **Initiate Individual Source Processes:** The master process reads the configuration data and initiates the landing and staging processes for each data source. This ensures that each source is handled independently and asynchronously.
     - **Configuration Data Interaction:** The master process interacts with a configuration table that holds all the necessary configurations. This makes the framework no-code, meaning changes can be made by updating configurations rather than altering the code.
2. **Landing/Staging Individual Source Processes (Child Processes)**
   - **Role:** Each child process is responsible for executing the data loading for a specific data source.
   - **Steps:**
     - **Landing Process:** Copy data from the gateway server to the temporary HDFS directory for the specific source.
     - **Staging Process:** Create a temporary external table on the temporary HDFS directory and insert data into the final ORC table.

4.1.2.1 Benefits of This Architecture

1. **Asynchronous Processing:**
    - By separating the master and child processes, data loading tasks can be executed asynchronously. This reduces the overall processing time as multiple data sources can be processed in parallel.
2. **Scalability:**
    - The architecture is highly scalable. New data sources can be added by simply updating the configuration table, without needing to modify the code.
3. **Modularity:**
    - Each child process operates independently, making the system modular. This improves maintainability and allows for easier troubleshooting and updates.
4. **Efficiency:**
    - The master process ensures that all child processes are initiated and managed efficiently. This reduces the risk of bottlenecks and improves resource utilization.
5. **Flexibility:**
    - The no-code approach allows for greater flexibility. Changes to the data loading process can be made quickly by updating the configuration data.

4.1.2.2 Flow of Processes

- **Master Process:** Reads configuration data and initiates individual source processes.
- **Child Processes:** Each child process handles the landing and staging for a specific data source, running independently and in parallel with other child processes.

This architecture is designed to improve the efficiency and manageability of data ingestion and processing, especially when dealing with multiple data sources.

4.1.3 Child Landing Batch Process

The Child Landing Batch Process is a crucial component of the proposed data ingestion architecture, designed to handle the initial stages of data processing independently for each data source. This process involves dynamically creating temporary and backup directories, ensuring that data is securely stored and readily available for further processing. By managing data through distinct stages—backup, temporary, and reject directories—the Child Landing Batch Process ensures efficient handling, logging, and storage of data. This modular approach not only enhances the flexibility and scalability of the data ingestion framework but also improves

resource utilization and reduces processing time by allowing parallel execution of tasks. Figure 4.3 outlines the Child Landing Batch Process.

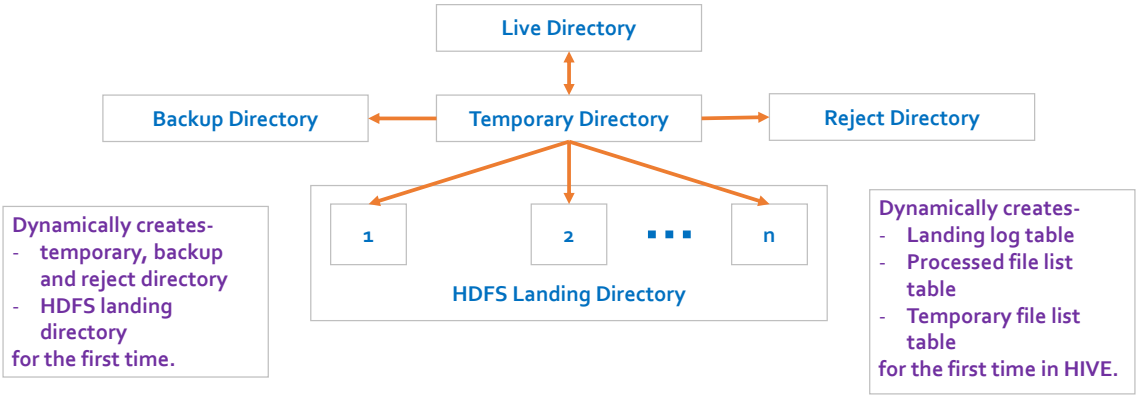

Figure 4.3: Child Landing Batch Process

1. **Temporary Directory**
   - **Function:** This is the initial stage where data is first stored.
   - **Steps:**
     - **Data Ingestion:** Data is ingested into the temporary directory.
     - **Processing Steps (1 to n):** The data undergoes various processing steps such as validation, transformation, and preparation for final storage.
2. **HDFS Landing Directory**
   - **Function:** After processing in the temporary directory, data is copied to the HDFS landing directory.
   - **Purpose:** Acts as the central hub for storing processed data before it is moved to the final destinations.
3. **Backup Directory**
   - **Function:** After data is copied to the HDFS landing directory, it is also backed up.
   - **Purpose:** Ensures that there is a secure backup of the data for recovery and integrity purposes.
4. **Reject Directory**
   - **Function:** Any data that fails validation or processing steps is moved to the reject directory.
   - **Purpose:** Manages rejected data by creating log tables and storing any data that doesn't meet the criteria. This helps in tracking and managing errors or discrepancies in the data.

4.1.3.1 Flow of Processes

1. **Data is first stored in the Temporary Directory:** This is where initial data ingestion and processing occur.
2. **Data is copied to the HDFS Landing Directory:** After processing, data is moved to the HDFS landing directory for centralized storage.
3. **Data is backed up in the Backup Directory:** Ensures data integrity and recovery by creating a backup.
4. **Rejected data is moved to the Reject Directory:** Manages and logs any data that doesn't meet the criteria.

4.1.3.2 Dynamic Creation of Directories and Tables

- **Temporary Directory:** The temporary directory is dynamically created to handle the initial data ingestion and processing steps. This ensures that the system can adapt to varying data loads and requirements without manual intervention.
- **HDFS Landing Directory:** The HDFS landing directory is dynamically created to store data after it has been processed in the temporary directory. This ensures that there is a centralized and organized location for all processed data, facilitating efficient data management and retrieval.
- **Backup Directory:** The backup directory is dynamically created for the first time to ensure that there is a secure backup of the data.
- **Reject Directory:** The reject directory is dynamically created to manage rejected data.
- **Landing Log Tables:** Tables that log the details of the data landing process.
- **Processed File List Tables:** Tables that keep track of files that have been processed.
- **Temporary File List Tables:** Tables that list temporary files created during the process.

This dynamic creation approach ensures that the system is flexible and can handle varying data loads efficiently. It also simplifies the management and maintenance of the data ingestion process by automating the creation of necessary directories and tables. This architecture ensures a systematic and efficient approach to data ingestion, processing, and management, with clear stages for temporary storage, backup, and rejection handling.

4.1.4 Child Staging Batch Process

The Child Staging Batch Process is a critical component of the proposed data ingestion architecture, designed to efficiently manage and process data through various stages. This process involves dynamically creating staging raw tables and a staging live table in Hive, ensuring that data from multiple sources is systematically organized and prepared for analysis. Data initially stored in the HDFS landing directories is transferred to corresponding staging raw tables, and then consolidated into a staging live table through dynamically generated selection queries. This structured approach not only enhances data management and processing efficiency but also ensures scalability and flexibility in handling large volumes of data. Figure 4.4 outlines the Child Landing Batch Process.

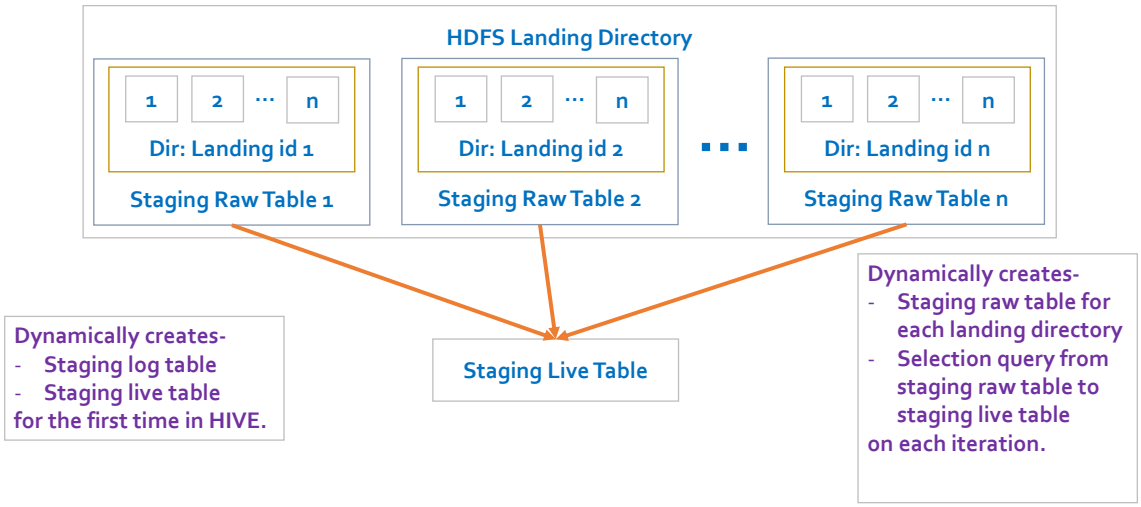

Figure 4.4: Child Staging Batch Process

1. **HDFS Landing Directory**
    - **Function:** Acts as the initial storage location for data after it has been processed in the landing batch process.
    - **Components:** Multiple directories labeled as "Dir: Landing id 1," "Dir: Landing id 2," up to "Dir: Landing id n," indicating that data from various sources is stored here.
2. **Staging Raw Tables**
    - **Function:** Each landing directory has a corresponding staging raw table in Hive.
    - **Dynamic Creation:** These staging raw tables are dynamically created for the first time in Hive to store raw data from the landing directories.
3. **Staging Live Table**
    - **Function:** Consolidates data from the staging raw tables into a single staging live table.
    - **Process:** Data is selected from each staging raw table and inserted into the staging live table on each iteration.

- **Dynamic Creation:** The staging live table is dynamically created for the first time in Hive.

4. **Staging Log Table**
    - **Function:** Logs the details of the staging process.
    - **Dynamic Creation:** The staging log table is dynamically created for the first time in Hive to track the staging process.
5. **Selection Query**
    - **Function:** Moves data from staging raw tables to the staging live table.
    - **Dynamic Creation:** The selection query is dynamically created for each iteration to ensure that data is accurately moved from the staging raw tables to the staging live table.

4.1.4.1 Flow of Processes

1. **Data is stored in the HDFS Landing Directory:** Data from various sources is initially stored in the respective landing directories.
2. **Data is moved to Staging Raw Tables:** Each landing directory's data is transferred to its corresponding staging raw table.
3. **Data is consolidated into the Staging Live Table:** Data from the staging raw tables is selected and inserted into the staging live table on each iteration.

This architecture ensures efficient data staging by organizing data through multiple stages, from landing directories to raw tables, and finally to a live table, facilitating better data management and processing.

4.1.5 Required Configuration for Data Loading

In the proposed architecture for data loading, a set of essential configurations ensures efficient and accurate ingestion of data. These parameters define how files are picked up, processed, and distributed within the system. From specifying source directories and file patterns to determining parallel processes and minimum file requirements, each configuration plays a crucial role in maintaining smooth data flow. Let's explore these settings in detail to understand their significance in the data loading process.

1. **Source id:**
    - This parameter serves as a small, unique identifier without spaces. It likely corresponds to a specific data source or category. For instance, the example "noaa_hourly" could represent hourly weather data from the National Oceanic and Atmospheric Administration (NOAA).
    - It's essential for organizing and managing data, especially when dealing with multiple sources.

2. **Server ip:**

    - The "Server ip" refers to the IP address of the server responsible for file distribution and executing batch processes. It's the gateway server.
    - The gateway server plays a crucial role in handling data transfers, processing, and distribution.

3. **Live directory:**

    - The "Live directory" specifies the physical directory on the gateway server where files are actively processed.
    - For example, "/landing_batch_process/noaa_hourly/data/" indicates the location where incoming files related to the "noaa_hourly" source are stored.

4. **File pattern:**

    - This parameter defines the pattern used to identify source files. The wildcard character "*" can be used to match any characters.
    - For instance, "*.csv" would include all CSV files within the specified directory.
    - This setting ensures that the system processes the correct file types.

5. **Parallel process:**

    - "Parallel process" indicates the number of concurrent processes required for the landing batch process.
    - Typically, it's set to 1 unless performance issues (such as slowness) are observed.

6. **Maximum files per landing process:**

    - Determines the maximum file processing capacity on each iteration of the landing batch process.
    - Values can be 50, 100, 500, 1000, or 5000.

7. **Threads per process:**

    - These threads are responsible for loading files into the Hadoop Distributed File System (HDFS) during the landing batch process.
    - The default value is 5, but adjustments may be necessary based on system performance.

8. **Minimum files requirement:**

    - The minimum number of files needed to initiate the landing batch process.
    - The specific requirement depends on the data source. For example, it could be 10, 20, 100, 500, or 1000 files.

9. **Landing status:**
    - Indicates whether the landing batch process is active or inactive.
    - Values: 0 (inactive) or 1 (active).
10. **Maximum landing per staging:**
    - Sets the maximum number of landing batch process data to select for each iteration.
    - Depends on the data source (e.g., 1, 2, 5, or 10).
11. **Fields name:**
    - Refers to source field names separated by commas.
    - Example: "field_name_1, field_name_2, field_name_3."
12. **Field delimiter:**
    - Specifies the character that separates each column in the data.
    - For example, a comma (",") is commonly used.
13. **Enclosed by:**
    - If fields are enclosed by any characters (e.g., quotes), this field provides that information; otherwise, it remains blank.
14. **File date identifier:**
    - Describes the logic for identifying the file date from the file name.
    - Used for extraction in Hive; logic may differ for different sources (e.g., "yyyymmdd").
15. **Staging status:**
    - Indicates whether the staging batch process is active or inactive.
    - Values: 0 (inactive) or 1 (active).

These configurations play a crucial role in setting up data loading processes within the proposed architecture. Adjusting these parameters correctly ensures efficient data ingestion and management.

4.1.6 One Single INSERT Query to Start Data Loading

Before starting the data loading process, an INSERT query is used to insert configuration data into the configuration table. This query is essential as it triggers the data loading process according to the proposed architecture, ensuring that all configurations are correctly set for seamless data ingestion into the target tables.

```
insert into configuration_table_name (
    source_id, server_ip, live_dir, file_pattern,
       parallel_process, max_files_per_iter,
       threads_per_process, min_files_requirement,
       landing_status, max_landing_per_staging, fields_name,
       field_delim, enclosed_by, file_date_identifier,
       staging_status
)
values(
    'sample_source_id', '192.168.1.1', '/sample/path/to/data/
       ', '*.csv', 1, 500, 5, 100, 1, 5, 'field1, field2,
       field3, ..., ',', '"', 'file_date_identifier logic', 1
);
```

4.2 Experimental Environment and Assumptions

In this study, the performance of the proposed asynchronous and parallel data loading architecture was evaluated in an idealized experimental environment. The focus was on measuring the efficiency of the Landing Batch and Staging Batch processes under controlled conditions. Key factors such as network congestion, bandwidth variability, and hardware limitations were deliberately excluded from the experimental setup.

While excluding these factors helped in providing a clearer assessment of the architecture's performance gains, it is important to recognize that real-world environments often face constraints such as network latency, data transfer bandwidth, and processing bottlenecks. These factors could influence the actual performance of the proposed architecture, and future experiments should aim to evaluate its robustness under more variable network conditions.

Chapter 5

Implementation

5.1 Dataset

A detailed overview of the dataset used in this study's implementation phase is provided. The dataset is crucial in validating the proposed data ingestion architecture and serves as a real-world example of large-scale data processing.

5.1.1 Dataset Descriptions

This study utilizes three distinct datasets to evaluate the effectiveness of the proposed architecture. These datasets vary in size, complexity, and structure, representing real-world challenges in big data processing.

Dataset 1: Global Hourly Weather Observations
The first dataset is the Global Hourly Weather Observations dataset, providing comprehensive hourly weather data from multiple global weather stations, available through the National Centers for Environmental Information (NCEI).

- **Source Name:** Global Hourly Weather Observations
- **Source URL:** https://www.ncei.noaa.gov/data/global-hourly/archive/csv/
- **Dataset Year:** 1989
- **Size:** 10.78 GB
- **File Count:** 10,840
- **Column Count:** 54
- **Record Count:** 36,290,914

Dataset 2: CERT Insider Threat Detection Research
The second dataset comes from CERT Insider Threat Detection Research and focuses on security logs related to insider threats. Only the http.csv file is considered for this experiment, providing large-scale network data.

- **Source Name:** CERT Insider Threat Detection Research

- **Source URL:** https://www.kaggle.com/datasets/mrajaxnp/cert-insider-threat-detection-research?select=http.csv
- **Size:** 20.67 GB
- **File Count:** 20,653
- **Column Count:** 7
- **Record Count:** 28,706,668

Dataset 3: Stock Market Data - Nifty 100 Stocks Data
The third dataset contains stock market data of 101 companies, sampled at 5-minute intervals, spanning from January 2015 to February 2022.

- **Source Name:** Stock Market Data - Nifty 100 Stocks Data
- **Source URL:** https://www.kaggle.com/datasets/debashis74017/stock-market-data-nifty-50-stocks-1-min-data?resource=download
- **Size:** 61.60 GB
- **File Count:** 63,095
- **Column Count:** 60
- **Record Count:** 63,944,474

5.1.2 Dataset Composition

The three datasets represent a wide range of data structures and characteristics, each serving as a unique test case for evaluating the performance and scalability of the proposed asynchronous and parallel architecture. This diversity in data composition allows for a comprehensive assessment of how the architecture handles different types of workloads and data challenges.

1. **Dataset 1: Global Hourly Weather Observations (NOAA)**
    - **Structure:** This dataset consists of structured time-series data, with each record representing hourly weather observations from various weather stations around the globe.
    - **Key Attributes:** The dataset includes 54 columns, capturing a broad range of meteorological parameters such as temperature, humidity, wind speed, and precipitation. Additionally, each record is tagged with metadata like station IDs, timestamps, and geographical coordinates.
    - **Challenges:**
        - Large file count (10,840) and relatively large dataset size (10.78 GB).
        - Handling time-series data with sequential dependencies, making it ideal for testing partitioning strategies based on temporal data (e.g., year and station ID).

- Requires efficient compression and query optimization for time-based retrieval in Hive.

- **Use Case:** Evaluates the architecture's ability to handle structured, multi-dimensional data with temporal components, a common requirement in meteorological and environmental analytics.

2. **Dataset 2: CERT Insider Threat Detection Research**

   - **Structure:** This dataset consists of log data related to insider threats, with HTTP traffic records forming the basis of the analysis. The data is semi-structured, with seven columns containing key information such as timestamps, source/destination IP addresses, and HTTP methods.
   - **Key Attributes:** 7 columns, which is considerably fewer than the other datasets, but a massive record count (28.7 million records) and file count (20,653).
   - **Challenges:**
     - Managing millions of small files, which can cause performance bottlenecks during file ingestion in Hadoop HDFS.
     - High data volume requires a parallel processing approach to handle the large number of records efficiently.
   - **Use Case:** Provides a real-world scenario for testing how the architecture handles cybersecurity log data, which is critical for organizations focused on monitoring insider threats and abnormal user behavior. This dataset is ideal for testing large-scale log analysis workflows with anonymization requirements.

3. **Dataset 3: Stock Market Data - Nifty 100 Stocks Data**

   - **Structure:** The dataset contains minute-level stock market data for 101 companies over a 7-year period, with each record representing stock prices and related metrics at 5-minute intervals. The data is highly structured and time-series-based.
   - **Key Attributes:**
     - A massive dataset with 63.9 million records and 60 columns, capturing details such as open, high, low, close (OHLC) prices, trading volume, and other financial indicators.
     - The data spans a long time period (2015-2022), making temporal partitioning critical for efficient query performance.
   - **Challenges:**
     - Handling extremely large files (63,095 files), which are spread across multiple stock symbols and time periods.
     - Ensuring efficient data partitioning based on stock symbol and date, especially given the large number of columns, which can lead to performance challenges during data insertion and retrieval.
     - Maintaining data integrity while processing large amounts of historical financial data across multiple companies.

- **Use Case:** This dataset allows the proposed architecture to be tested in a financial services context, where large volumes of high-frequency stock market data need to be processed and analyzed. It tests the architecture's ability to support time-series analysis and quick query retrieval, which is essential for financial analytics and trading systems.

5.1.3 Data Format and Characteristics

The dataset is provided in CSV format, which, while straightforward and easy to read, presents certain limitations when dealing with large-scale data. CSV files are not inherently optimized for performance, lack built-in support for schema evolution, and are not space-efficient. As part of the ingestion process, these files can be converted into more efficient formats such as ORC or Parquet, which support advanced features like compression, indexing, and optimized columnar storage, significantly enhancing performance and storage efficiency.

5.1.4 Relevance to the Study

The inclusion of three diverse datasets in this study is fundamental to providing a comprehensive evaluation of the proposed asynchronous and parallel architecture. Each dataset introduces unique characteristics and challenges, allowing for a thorough assessment of the architecture's flexibility, scalability, and performance under varying real-world conditions. By testing across these diverse data types, this study demonstrates the architecture's ability to handle multiple use cases typically encountered in big data environments.

1. **Handling of Time-Series Data**

   - **Weather Data (NOAA) and Stock Market Data (Nifty 100):** Both the weather data and stock market data are inherently time-series datasets, which require efficient handling of large volumes of temporally ordered data. These datasets pose challenges related to time-based partitioning, sequential dependencies, and efficient retrieval of records over specific periods.
     - **Relevance to the Study:** Time-series data is common in a variety of fields, including meteorology, finance, and IoT (Internet of Things). The architecture's performance in processing these datasets showcases its ability to efficiently ingest, partition, and query time-dependent data, an essential feature for systems dealing with real-time analytics or historical data analysis.
     - **Parallel Processing Benefits:** By employing parallel processing techniques, the architecture can optimize data ingestion and retrieval, significantly reducing latency and improving throughput for large time-series datasets. This is particularly beneficial in scenarios like financial market analysis, where real-time data processing is critical for decision-making.

2. **Event Logs and Cybersecurity Use Case**

   - **CERT Insider Threat Dataset:**
     The CERT dataset consists of HTTP traffic logs, a form of event-based log data typically found in cybersecurity, network monitoring, and user behavior analysis. Log data can be highly unstructured and voluminous, often requiring sophisticated techniques for real-time processing, filtering, and analysis.
     - **Relevance to the Study:** Event log data, such as security logs or system audits, are vital in industries like cybersecurity, where real-time detection of threats and anomalies is crucial. Testing the architecture on this type of data ensures it can handle high-velocity log streams and large volumes of semi-structured data while supporting real-time or near-real-time monitoring.

3. **Scalability Across Large Datasets**

   - The three datasets combined represent over 95 million records and nearly 100 GB of data, spread across over 100,000 files. Each dataset poses its own set of challenges in terms of data volume, file count, and record structure.
     - **Relevance to the Study:** Scalability is one of the key criteria for evaluating big data architectures. Testing the architecture with such large datasets provides insights into how well it handles large-scale data ingestion, storage, and retrieval. Additionally, it evaluates the efficiency of the architecture in terms of resource utilization (CPU, memory, I/O) and throughput, especially under high data loads.

4. **Multiple Data Structures and Use Cases**

   - **Structured Data (Stock Market and Weather Data):** These datasets are highly structured, with clearly defined columns and predictable data types. Handling structured data effectively is essential for applications in finance, meteorology, supply chain analytics, and more.
   - **Semi-Structured Data (CERT Logs):** The CERT dataset introduces a semi-structured data format, commonly encountered in logs, click-streams, and other forms of machine-generated data. Processing such data efficiently is crucial for industries dealing with event monitoring, auditing, and cybersecurity.
     - **Relevance to the Study:** Testing across both structured and semi-structured data formats allows the study to demonstrate the versatility of the architecture. It shows how the system adapts to different data formats, ingesting structured data efficiently while also being capable of processing less organized, semi-structured logs. This is especially important for enterprises that deal with hybrid environments, requiring flexibility to process various types of data simultaneously.

5. **Real-World Conditions**

   - The diverse datasets reflect real-world scenarios faced by businesses and organizations across industries. Meteorological data (NOAA) simulates weather forecasting or environmental monitoring applications. Financial stock market data (Nifty 100) mirrors the needs of financial services for real-time analysis and decision-making. The CERT log data is representative of cybersecurity monitoring, where event logs need to be processed in real time to detect potential insider threats.
     - **Relevance to the Study:** Testing the architecture under these varied conditions validates its applicability to a wide range of real-world use cases. By covering industries like meteorology, finance, and cybersecurity, the study proves that the architecture can deliver reliable, scalable, and efficient performance across multiple sectors.

5.1.5 Challenges and Solutions

The ingestion, processing, and analysis of the three diverse datasets used in this study present several technical challenges. These challenges are tied to the size, structure, and performance demands of the datasets. However, the proposed asynchronous and parallel architecture addresses these obstacles through a series of optimized strategies, ensuring that the system remains efficient, scalable, and adaptable across various data environments.

1. **High Volume: Managing Large Files and Data Sizes**

   - **Challenge:**
     - The datasets used in the study amount to over 90 GB in size and consist of over 100,000 files. Managing such large volumes of data is inherently challenging due to I/O bottlenecks, storage requirements, and the need for efficient data retrieval mechanisms.
     - Processing large datasets without overwhelming system resources (CPU, memory, disk) requires careful optimization, particularly when dealing with multiple files simultaneously.
   - **Solution:**
     - **Parallel Data Ingestion:** The architecture uses a parallel data ingestion process, where files are loaded into the system simultaneously, reducing the overall time required to handle large datasets. This is particularly useful when dealing with thousands of files, as the system can process them concurrently instead of sequentially.
     - **Batch Processing:** The architecture groups files into batches for processing, which allows it to manage high-volume data more efficiently. Batch processing reduces the load on system resources and ensures that data is ingested and processed without overwhelming the cluster.
     - **Distributed File System (HDFS):** Leveraging the Hadoop Distributed File System (HDFS) ensures scalability by distributing files

across multiple nodes. This helps balance the storage requirements and prevents individual nodes from becoming bottlenecks during data ingestion and processing.

2. **Data Structure Complexity: Handling Different Schemas and Data Types**

   - **Challenge:**
     - The datasets exhibit different structural complexities, with varying column counts, data types, and levels of structure. The NOAA weather dataset contains 36 columns, the CERT insider threat dataset has 7 columns, and the Nifty 100 stock market dataset consists of 60 columns. Each of these datasets requires unique schema definitions for processing and storage.
     - In addition to varying column counts, the datasets involve different data types, including numeric data, timestamps, categorical variables, and possibly unstructured text (in event logs). These differences make schema handling and data transformation complex.
   - **Solution:**
     - **Flexible Schema Management:** The architecture is designed to be schema-agnostic, meaning it can handle datasets with varying structures without requiring significant reconfiguration. The system dynamically adapts to each dataset's schema during ingestion, ensuring that different column counts and data types are managed efficiently.
     - **Schema Evolution Support:** In environments where data structures may change over time (e.g., new columns being added or existing ones being modified), the architecture supports schema evolution. This is particularly useful for continuously updated datasets like stock market or weather data, where the data format might evolve.
     - **Data Format Optimization:** The system converts raw data into optimized formats, such as ORC or Parquet, that are well-suited for big data processing in distributed environments. These columnar storage formats provide faster access to specific columns, reduce storage costs through compression, and enable better schema handling, especially for wide tables like the stock market data with 60 columns.

3. **Performance Optimization: Efficient Data Loading, Querying, and Parallel Processing**

   - **Challenge:**
     - Ingesting, transforming, and querying large datasets in a timely manner is critical for big data applications. Performance bottlenecks can occur during data loading, format conversion, and querying, especially when the system needs to handle millions of records (nearly 30 million in the CERT dataset, and over 60 million in the stock market dataset).

- High query performance is also necessary to ensure that data scientists and analysts can extract insights from these datasets without experiencing long delays, especially in real-time or near-real-time environments (e.g., financial market data analysis).

- **Solution:**
  - **Parallel Processing Architecture:** The core feature of the proposed architecture is the parallelization of processes, both during the data ingestion phase (Landing Batch Process) and the transformation phase (Staging Batch Process). By executing multiple processes concurrently, the architecture reduces processing times and balances the workload across the cluster.
    - **Example:** During the landing process, two parallel processes were used to handle data ingestion, resulting in significant reductions in processing time compared to a single-process approach.
  - **Data Format Conversion and Compression:** Converting data into efficient storage formats like ORC (Optimized Row Columnar) ensures faster queries and reduced I/O overhead. These formats are designed to store large datasets compactly while providing high performance for query execution, particularly for columnar operations.

4. **Balancing Resource Utilization and System Efficiency**

- **Challenge:**
  - Balancing the system's resource utilization (CPU, memory, disk I/O) while processing large datasets is critical to avoid bottlenecks that could degrade performance. Inefficient resource management can lead to excessive memory usage, disk thrashing, and reduced overall throughput.

- **Solution:**
  - **Resource Allocation and Tuning:** The architecture includes resource management strategies to ensure that CPU, memory, and I/O are utilized efficiently. Resource tuning configurations, such as adjusting the number of parallel processes or the size of data chunks, allow the system to optimize resource utilization based on the workload.
  - **Distributed Processing Framework (YARN):** The architecture utilizes YARN (Yet Another Resource Negotiator) for resource management in the Hadoop ecosystem. YARN ensures that tasks are distributed across the cluster in a way that optimally uses available resources, balancing load and preventing any one node from becoming a bottleneck.

5.2 Environment Setup

This section provides details on the environment setup used for processing the three diverse datasets: the Global Hourly Weather Observations, the CERT Insider Threat Detection Research, and the Stock Market Data - Nifty 100 Stocks datasets. With a combined total of over 90 GB of data distributed across approximately 94,000 files, the environment needed to be robust, scalable, and capable of handling complex, high-volume data processing.
The EMR cluster was specifically configured to handle the high data volume and complexity, providing an efficient and scalable solution for big data processing. The cluster was integrated with Apache Hive and Apache Spark to manage the data pipeline and facilitate analytics.

5.2.1 Amazon EMR

Amazon EMR is a managed cluster platform that simplifies running big data frameworks such as Apache Hadoop, Spark, Hive, and HDFS. It is well-suited for processing and analyzing vast datasets, providing scalability, flexibility, and integration with other AWS services like Amazon S3. For this work, Amazon EMR was selected for its ability to process large datasets efficiently and its seamless integration with the data storage and analytics ecosystem.

5.2.2 EMR Cluster Configuration

The EMR cluster was configured with optimized settings to manage the batch processing of the dataset and support large-scale data analytics. The following are the key configurations and specifications of the EMR cluster:

- **Cluster Version:** Amazon EMR version 6.x, which includes Apache Hadoop, Apache Hive, and Apache Spark.
- **Instance Types:** The cluster was set up with different node types, each serving a specific role in the data processing pipeline:
  - **Master Node:** 1 instance (m5.xlarge) with 16 GB RAM, 4 vCPUs – responsible for managing the cluster, coordinating tasks, and resource allocation.
  - **Core Nodes:** 4 instances (m5.xlarge), each with 16 GB RAM and 4 vCPUs – responsible for running the Hadoop Distributed File System (HDFS) and executing data processing tasks.
  - **Task Nodes:** 2 instances (m5.large), each with 8 GB RAM and 2 vCPUs – dedicated to processing tasks using Spark and Hive, providing additional computational power to the cluster.
- **Storage Configuration:** Each Core node was equipped with Elastic Block Store (EBS) volumes to provide ample disk space for intermediate data processing and HDFS storage. HDFS served as the main storage system within the cluster, allowing for high-throughput data access.

5.2.3 Integration with Apache Hive and Apache Spark

The cluster was configured with Apache Hive and Apache Spark, key components in the big data ecosystem that allow for efficient data processing, querying, and analytics:

- **Apache Hive:** Hive was installed on the EMR cluster to facilitate SQL-like querying on large datasets stored in HDFS. Hive provides a structured query language (HiveQL) that is similar to SQL, making it easier to work with large-scale data. The Hive environment was configured to interact seamlessly with HDFS, allowing data to be accessed, transformed, and managed within the cluster.
- **Apache Spark:** Spark was included in the cluster configuration to provide fast, in-memory data processing capabilities. Spark's data processing speed and support for complex transformations made it ideal for handling the batch processing of the dataset. Spark jobs were executed on the Task nodes, leveraging the parallel processing power of the cluster to process data efficiently.

The EMR cluster setup provided a powerful and scalable environment to process and analyze all three datasets: Global Hourly Weather Observations, CERT Insider Threat Detection Research, and Stock Market Data - Nifty 100 Stocks. The integration of Apache Hive and Apache Spark within the EMR ecosystem enabled efficient data handling, from ingesting raw data to performing complex queries and analytics across a wide variety of data structures and formats.

The environment was carefully configured to optimize resource utilization for the diverse datasets, ensuring efficient parallel processing of over 94,000 files with a combined size of more than 90 GB. Key configurations focused on handling the distinct challenges of each dataset, such as processing high-volume time-series data, managing large event logs, and working with complex financial data structures. Security was also prioritized, particularly for sensitive information in the CERT dataset.

This setup provided a robust, scalable platform that aligned with the architecture's objectives, ensuring that each dataset could be processed efficiently while supporting advanced analytics and high-performance data transformations.

5.2.4 Gateway/Staging/Edge Node

1. **Purpose of Gateway/Staging/Edge Node:**
    - The Gateway or Edge Node acts as an intermediary between the external data sources (like Amazon S3) and the EMR cluster. It is responsible for tasks such as data ingestion, preprocessing, and staging before the data is loaded into HDFS.
    - This node helps in decoupling data loading tasks from the core and master nodes, ensuring that the main nodes focus on data processing and not on data transfer.

2. **Node Selection for Gateway/Staging:**

    - A Task Node was dedicated solely as an Edge Node. This node won't store HDFS data permanently and won't contribute to the processing workload, allowing it to focus entirely on data staging and loading tasks.

3. **Configuration for Edge Node:**

    - **Instance Type:** A Task Node (m5.xlarge) with 16 GB RAM, 4 vCPUs was chosen.
    - **Storage:** Attached sufficient EBS (Elastic Block Store) volumes to provide temporary storage for incoming data before it is transferred to HDFS.
    - **Network Setup:** Ensured that the Edge Node has proper access to external data sources (like Amazon S3) and internal cluster components (HDFS, Hive).

5.3 Data Preprocessing

The preprocessing of data is a critical step in preparing the diverse datasets for efficient processing and analysis in the EMR environment. This project uses three datasets: Global Hourly Weather Observations, CERT Insider Threat Detection, and Stock Market Data - Nifty 100 Stocks. Each dataset requires specific preprocessing steps to ensure it is ready for ingestion and analysis. Below are the detailed steps involved in the data preprocessing phase:

1. **Data Acquisition:**

    - **Global Hourly Weather Observations:** This dataset was downloaded from the NOAA website in compressed format (ZIP). The initial download was approximately 1 GB, containing historical weather data in CSV format.
    - **CERT Insider Threat Detection:** The http.csv file, around 20.67 GB in size, was downloaded from Kaggle, containing security event logs related to insider threats.
    - **Stock Market Data - Nifty 100 Stocks:** This dataset, sized at 61.59 GB, was downloaded from Kaggle, consisting of time-series data in CSV format for stock prices with a 5-minute interval from January 2015 to February 2022.

2. **Uploading Data to Amazon S3:** After downloading, the compressed files for each dataset were uploaded to an Amazon S3 bucket. Amazon S3 was used as the primary storage location, providing scalable, durable, and secure data storage. It facilitated easy management, ensuring all datasets were ready for further processing.

3. **Transferring Data to the Edge Node:** Each dataset was then transferred from S3 to the Edge Node of the EMR cluster. This was done using AWS CLI commands such as aws s3 cp to ensure efficient data transfer without

overwhelming the core nodes responsible for HDFS storage and processing.

```
aws s3 cp s3://<bucket-name>/<file-name>.zip /local/path/
   on/edge-node/
```

This approach ensures that data movement is streamlined and does not directly interfere with the core nodes responsible for HDFS storage and processing.

4. **Extracting the Zipped File:** On the Edge Node, the compressed files were extracted using standard UNIX commands (unzip or tar). Each dataset had varying sizes and record counts:

   - **Global Hourly Weather Observations:** Expanded into 10,840 individual CSV files, totaling approximately 10.78 GB.
   - **CERT Insider Threat Detection:** Expanded into over 20,653 CSV files, with a total size of 20.67 GB.
   - **Stock Market Data:** Expanded into 63,095 CSV files, with a total size of 61.59 GB.

   Example extraction command:

```
unzip /local/path/on/edge-node/<file-name>.zip -d /local/
   path/on/edge-node/extracted/
```

   The extraction process is essential for converting the single compressed file into multiple readable CSV files, ready for further ingestion into the data processing pipeline.

5. **Data Verification and Integrity Check:** After extraction, integrity checks were performed to ensure that files were correctly extracted and no data was corrupted. File counts and total sizes were verified using commands such as ls -l — wc -l and du -sh to match the expected values:

   - **Global Hourly Weather Observations:** 10,840 files, 10.78 GB.
   - **CERT Insider Threat Detection:** 20,653 files, 20.67 GB.
   - **Stock Market Data:** 63,095 files, 61.59 GB.

6. **Directory Structuring:** The extracted files for each dataset were organized into structured directories on the Edge Node, making them easily accessible for subsequent processing and ingestion. Directory naming conventions were maintained for each dataset to facilitate the smooth loading of files into HDFS, ensuring that different batches could be identified and processed efficiently.

7. **Data Cleanup:** Temporary files and logs generated during the extraction and transfer processes were cleaned up to free up space on the Edge Node. This step was essential to ensure optimal performance during the data loading phase and to prevent unnecessary overhead in the system.

These preprocessing steps are critical in ensuring that the data is in the correct format and location for further processing within the data pipeline. By carefully managing the data extraction and transfer processes, the dataset is prepared efficiently, maintaining data integrity and performance throughout the pipeline.

5.4 Data Loading

The data loading process in this work is divided into two main sections: the Traditional Architecture and the Proposed Architecture. Each section consists of two phases: the Landing Batch Process and the Staging Batch Process. The goal is to evaluate the performance of both architectures by comparing execution times, standard deviation, and mean across multiple datasets. While the NOAA dataset was tested over 10 iterations for each step, the CERT Insider Threat Detection and Stock Market datasets were tested with only 1 iteration per step. Below is a detailed explanation of the implementation steps for each architecture.

5.4.1 Traditional Architecture

5.4.1.1 Landing Batch Process

- **Loading Files into HDFS:** In the traditional architecture, the Landing Batch Process sequentially loads the extracted data files from the edge node into HDFS. The files are processed one by one, without any parallelism, ensuring that each file is completely transferred before the next file is processed. This step applies to all three datasets.

- **Sequential Data Loading:** No parallel processing is employed, meaning each file is loaded into HDFS individually. This simple, straightforward approach guarantees correct order and data integrity but is relatively slow when compared to parallel loading techniques.

- **Performance Evaluation:** For the NOAA dataset, this process was executed 10 times to gather performance metrics such as average execution time, standard deviation, and mean values, which serve as a benchmark for comparing with the proposed architecture.
  For the CERT and Stock Market datasets, the process was executed only once due to the large size of the files and limited experimental scope. The performance metrics were recorded after the single iteration for these datasets.

5.4.1.2 Staging Batch Process

- **Inserting Data into Hive:** After the data is loaded into HDFS, the Staging Batch Process is responsible for inserting the data from HDFS into Hive tables. In the traditional approach, this is done sequentially, one file at a time.

- **Performance Repetition:** Similar to the Landing Batch Process, the staging process for the NOAA dataset was executed 10 times to capture average execution time, standard deviation, and mean values.
  For the CERT and Stock Market datasets, the process was performed only once, with the results recorded for comparison with the proposed architecture.

5.4.2 Proposed Architecture

The proposed architecture introduces parallel processing capabilities to improve the efficiency of the loading and staging processes. This architecture was tested under two configurations: (1) without parallelism and (2) with two parallel processes. The goal is to evaluate whether parallelism and asynchronous task handling can enhance the overall data processing performance.

5.4.2.1 Landing Batch Process with No Parallel

- **Asynchronous Data Loading:** This approach begins similarly to the traditional method, where data is loaded into HDFS from the edge node, but now under the proposed architecture framework. The extracted files are loaded into HDFS without any parallel threads, but the architecture allows for asynchronous management of tasks. This modification aims to minimize idle time and improve efficiency slightly even when operating without explicit parallelism.

- **Iteration and Evaluation:** For the NOAA dataset, this process was repeated 10 times to capture performance metrics such as average execution time, standard deviation, and mean values.
  For the CERT and Stock Market datasets, the process was executed only once, and metrics were recorded accordingly. The results from these single iterations provide an initial comparison point with the traditional approach and will help gauge improvements when parallelism is introduced.

5.4.2.2 Landing Batch Process with 2 Parallel

- **Parallel Data Loading:** To further enhance performance, this setup introduces parallelism, allowing two data streams to run concurrently, thereby reducing total data loading time. Two parallel processes are used to load data into HDFS simultaneously. This configuration significantly speeds up the process as files are processed concurrently, unlike the sequential approach used previously. This approach efficiently utilizes system resources, particularly when working with large volumes of data, reducing overall data loading time.

- **Performance Assessment:** For the NOAA dataset, this process was repeated 10 times to analyze the improvements in execution time, standard deviation, and mean values compared to the sequential approaches.
  For the CERT and Stock Market datasets, the process was only executed once. This single iteration still provided valuable insights into the potential benefits of parallelism when applied to high-volume, large-scale datasets.

5.4.2.3 Staging Batch Process

- **Asynchronous Data Insertion:** The final step in the proposed architecture is the Staging Batch Process, where data loaded into HDFS is inserted into Hive tables asynchronously. Unlike the traditional approach, the proposed staging process allows for the data insertion to be handled asynchronously, meaning data can be staged and loaded into Hive tables concurrently with

ongoing data loads. This non-blocking approach improves throughput and reduces the total time required for data to become available for analysis.

- **Repeated Execution for Performance Metrics:** For the NOAA dataset, this process was executed 10 times, capturing performance metrics such as execution time, standard deviation, and mean values to evaluate improvements compared to the traditional approach.
  For the CERT and Stock Market datasets, the process was executed once, and performance metrics were recorded after that single iteration.

Chapter 6

Result and Analysis

6.1 Data Size Analysis

The effectiveness of data storage formats and replication strategies plays a crucial role in optimizing storage utilization, especially when dealing with large-scale datasets in data lakes or data warehouses. We analyzed the impact of different data loading architectures and storage formats, focusing on the comparison between traditional raw file storage and the optimized ORC (Optimized Row Columnar) format across three datasets: NOAA Global Hourly Weather Observations, CERT Insider Threat Detection, and Stock Market data. The table below summarizes the impact of replication and storage formats on dataset size across all three datasets.

6.1.1 Traditional Data Loading with Raw Files

In the traditional data loading approach, the extracted files from the edge node are directly copied into HDFS without any optimization in the storage format. HDFS by default replicates each file three times to ensure fault tolerance and data redundancy. This replication drastically increases the storage requirements for raw files:

- **NOAA Weather Data:**
  - Initial File Size: 10.78 GB
  - Size After 3 Replications: 32.4 GB
- **CERT Insider Threat Data:**
  - Initial File Size: 20.67 GB
  - Size After 3 Replications: 62.01 GB
- **Stock Market Data:**
  - Initial File Size: 61.59 GB
  - Size After 3 Replications: 184.77 GB

As seen in Table 6.1, the replication factor of 3 results in a substantial increase in the storage space required. For instance, the NOAA dataset grows from 10.78 GB to 32.4 GB, and the CERT and Stock Market datasets expand similarly. This highlights the inefficiency of using raw file formats for large datasets, as the storage footprint significantly increases, leading to higher storage costs and resource consumption.

Dataset	Extracted File Size (GB)	Raw File Size (GB) After Replication	ORC File Size (GB) Before Replication	ORC File Size (GB) After Replication
NOAA Weather Data	10.78	32.4	0.8663	2.6
CERT Insider Threat Data	20.67	62.01	1.63	4.88
Stock Market Data	61.59	184.77	4.84	14.53

Table 6.1: Comparison of File Formats and Storage Sizes Across Datasets

6.1.2 Optimized Data Storage with ORC Format

To address the storage inefficiencies of raw files, the datasets were loaded into Hive tables using the ORC format. ORC is a highly optimized columnar storage format that provides efficient data compression, faster read/write performance, and is especially suited for analytical workloads. The table below illustrates the data size reduction when using ORC format:

- **NOAA Weather Data:**
  - ORC File Size Before Replication: 866.3 MB
  - ORC File Size After 3 Replications: 2,598.6 MB
- **CERT Insider Threat Data:**
  - ORC File Size Before Replication: 1.63 GB
  - ORC File Size After 3 Replications: 4.88 GB
- **Stock Market Data:**
  - ORC File Size Before Replication: 4.84 GB
  - ORC File Size After 3 Replications: 14.53 GB

By converting the raw data into ORC format, the storage size is significantly reduced. ORC format compresses the data and optimizes it for faster query performance due to its columnar structure. For example, the NOAA dataset shrinks to just 866.3 MB from its original size of 10.78 GB before replication, a substantial reduction in storage footprint.

6.1.3 Comparison of Storage Reduction

The transformation from raw file storage to ORC format demonstrates a significant reduction in storage requirements across all three datasets are shown in Table 6.2.

Dataset	Percentage Size Reduction	Size Reduction Factor (Times)
NOAA Weather Data	92.17%	12.77
CERT Insider Threat Data	92.13%	12.70
Stock Market Data	92.14%	12.72

Table 6.2: Storage Reduction with ORC Format Across Datasets

- **NOAA Weather Data:** The ORC format reduced the file size by **92.17%**, shrinking the dataset by a factor of **12.77 times**.

- **CERT Insider Threat Data:** The file size was reduced by **92.13%**, compressing the dataset **12.70 times**.
- **Stock Market Data:** ORC reduced the file size by **92.14%**, with a **12.72 times** reduction.

This analysis confirms the effectiveness of the ORC format in significantly reducing data size, even after HDFS replication. The storage space required for ORC files is far less than raw files, making ORC a highly efficient choice for large-scale datasets.

6.1.4 Implications for Data Warehousing

The results clearly show that ORC is an optimal choice for storing large-scale datasets in data warehousing environments. The drastic reduction in size not only saves storage space but also enhances the overall performance of data loading and querying processes. Smaller file sizes reduce disk I/O, enable faster read and write speeds, and lower resource consumption, which directly improves cost efficiency and scalability of the data warehouse.

Additionally, using ORC format tables in Hive allows for better integration with analytical tools and faster processing times, making it a preferred choice for large-scale data analytics platforms.

The results validate the superiority of ORC format over traditional raw file storage, especially in environments dealing with high data volumes and frequent access patterns. This substantial reduction in storage size and performance benefits makes ORC a strategic choice for optimizing data storage in modern data warehousing solutions. Selecting the right data format is crucial for balancing storage efficiency, performance, and cost in large-scale data processing systems.

6.2 Landing Batch Process with No Parallel

Landing Time (sec)	Normal Distribution
397	0.00359840
407	0.00525122
408	0.00543041
447	0.01101185
451	0.01108002
457	0.01092724
466	0.01015904
470	0.00963991
496	0.00507402
511	0.00276401

Table 6.3: Landing Batch Process with No Parallel - Normal Distribution Values for NOAA Dataset

We analyzed the performance of the landing batch process without parallelism for three datasets: NOAA Weather Data, CERT Insider Threat Data, and Stock Market Data. The landing times were measured across 10 iterations for the NOAA dataset, while the CERT and Stock datasets were measured once due to their larger size. The landing times were analyzed to determine the average, standard deviation, and normal distribution for the NOAA dataset, and key performance metrics for the other two datasets.

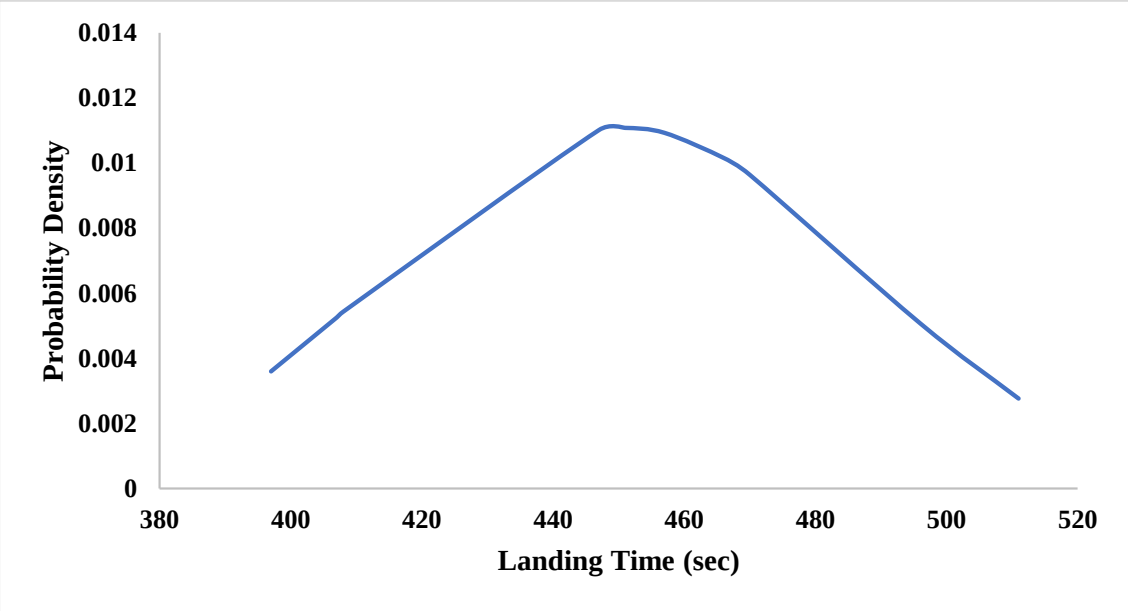

Figure 6.1: Normal Distribution of Landing Batch Process with No Parallel for NOAA Dataset

- **NOAA Weather Data:** The data collected from 10 iterations of the landing batch process for the NOAA Weather Data are shown in Table 6.3.
  - **Average Landing Time:** The average (mean) landing time was calculated to be **451 seconds**. This value represents the central tendency of the landing times across the 10 iterations.
  - **Standard Deviation:** The standard deviation of the landing times was **36.01 seconds**, indicating how much the landing times vary from the average.
  - **Normal Distribution Analysis:** The normal distribution values provided for each landing time indicate how frequently each landing time occurs relative to the others. The Figure 6.1 of the normal distribution shows a bell-shaped curve, which is characteristic of a normal distribution. The peak of the curve corresponds to the average landing time, and the spread of the curve is influenced by the standard deviation.
  - **Consistency of Landing Times:** The average landing time of **451 seconds** suggests that, on average, the landing batch process takes around 7.5 minutes to complete. The standard deviation of **36.01 seconds** indicates that most landing times fall within a range of approximately **415 to 487 seconds** (mean $\pm$ one standard deviation).

- **CERT Insider Threat Data:** For the CERT Insider Threat dataset, the landing batch process was executed without parallelism, and the landing time was recorded for a single run:
  - **Landing Time:** The recorded landing time for the CERT dataset was **861 seconds**, which is significantly higher than that of the NOAA dataset due to the larger file size (20.67 GB compared to 10.78 GB for NOAA).
  - **Execution Duration:** This landing time indicates that the CERT dataset took approximately **14.35 minutes** to complete the landing process.
- **Stock Market Data:** The Stock Market dataset, being the largest of the three, also went through the landing batch process without parallelism, with the following results:
  - **Landing Time:** The recorded landing time for the Stock dataset was **2,643 seconds**, reflecting the considerable size of this dataset (61.59 GB).
  - **Execution Duration:** This landing time translates to approximately **44 minutes**, demonstrating the impact of file size on landing time in a non-parallel architecture.
- **Summary of Landing Times (No Parallel):**
  - **NOAA Weather Data (10 Iterations):**
    - Average Landing Time: 451 seconds (7.5 minutes)
    - Standard Deviation: 36.01 seconds
  - **CERT Insider Threat Data (Single Run):**
    - Landing Time: 861 seconds (14.35 minutes)
  - **Stock Market Data (Single Run):**
    - Landing Time: 2,643 seconds (44 minutes)

  This analysis illustrates that, as dataset size increases, so too does the time required for the landing batch process, especially when no parallelism is employed. While the NOAA dataset exhibited a stable landing time with low variation, the larger CERT and Stock datasets showed a significant increase in time required for completion due to their larger file sizes.

6.3 Landing Batch Process with 2 Parallel

We analyzed the performance of the proposed asynchronous and parallel architecture for Hive, focusing on the landing batch process executed with 2 parallel processes. The data loading times were measured across 9 iterations for the NOAA dataset, while the CERT Insider Threat and Stock Market datasets were measured once due to their larger size. The results were analyzed to determine the distribution, average, standard deviation, and key performance metrics.

- **NOAA Weather Data:** The data collected from 9 iterations of the landing batch process with 2 parallel executions for the NOAA Weather Data are shown in Table 6.4.

Landing Time (sec)	Normal Distribution
376	0.003957647
391	0.020439397
398	0.028330066
401	0.029909463
403	0.030138070
407	0.028573784
410	0.025856269
412	0.023508814
426	0.006369608

Table 6.4: Landing Batch Process with 2 Parallel - Normal Distribution Values for NOAA Dataset

- **Average Landing Time:** The average (mean) landing time for NOAA was calculated to be **402.67 seconds**, representing the central tendency across 9 iterations.
- **Standard Deviation:** The standard deviation of the landing times was found to be **13.23 seconds**. This indicates the amount of variation or dispersion from the average landing time. A lower standard deviation suggests that the landing times are more consistent, while a higher standard deviation indicates greater variability.

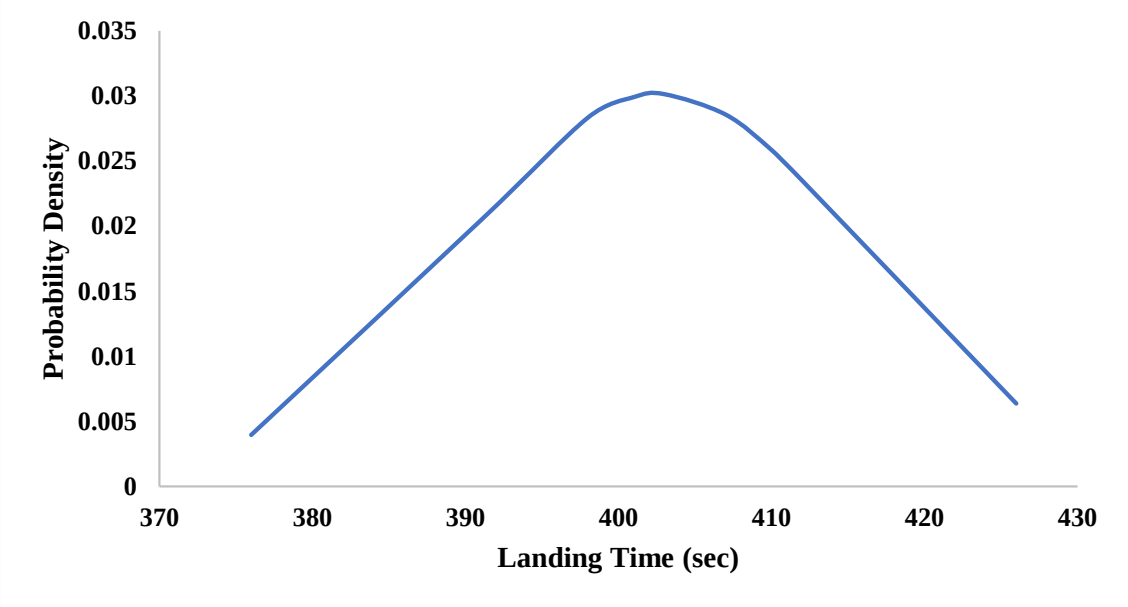

Figure 6.2: Normal Distribution of Landing Batch Process with 2 Parallel for NOAA Dataset

- **Normal Distribution Analysis:** The normal distribution values provided for each landing time indicate how frequently each landing time occurs relative to the others. The Figure 6.2 of the normal distribution shows a bell-shaped curve, which is characteristic of a normal distribution. The peak of the curve corresponds to the average landing time, and the spread of the curve is influenced by the standard deviation.

  - **Consistency of Landing Times:** The average landing time of **402.67 seconds** suggests that, on average, the landing batch process takes around 6.7 minutes to complete when executed with 2 parallel processes. The standard deviation of **13.23 seconds** indicates that most landing times fall within a range of approximately **389.44 to 415.90 seconds** (mean ± one standard deviation).

- **CERT Insider Threat Data:** For the CERT Insider Threat dataset, the landing batch process was executed with 2 parallel processes, and the landing time was recorded for a single run:
  - **Landing Time:** The recorded landing time for the CERT dataset was **763 seconds** (approximately 12.7 minutes), significantly faster than the non-parallel execution time of **861 seconds**.
  - **Execution Duration:** This reduction highlights the efficiency gains achieved through parallel execution, with a reduction of nearly 100 seconds compared to the non-parallel process.

- **Stock Market Data:** The Stock Market dataset, being the largest of the three, also went through the landing batch process with 2 parallel processes, with the following results:
  - **Landing Time:** The recorded landing time for the Stock dataset was **2,342 seconds** (approximately 39 minutes), compared to **2,643 seconds** in the non-parallel architecture.
  - **Execution Duration:** The parallel architecture reduced the landing time by about **301 seconds**, or approximately **5 minutes**, demonstrating the impact of parallel processing on large datasets.

- **Summary of Landing Times (2 Parallel):**
  - **NOAA Weather Data (9 Iterations):**
    - Average Landing Time: 402.67 seconds (6.7 minutes)
    - Standard Deviation: 13.23 seconds
  - **CERT Insider Threat Data (Single Run):**
    - Landing Time: 763 seconds (12.7 minutes)
  - **Stock Market Data (Single Run):**
    - Landing Time: 2,342 seconds (39 minutes)

6.4 Analysis of Both Landing Batch Process

Comparing the landing times with and without parallel execution across all datasets highlights the efficiency gains achieved through parallel processing are shown in Table 6.5.

- **NOAA Weather Data:** The average landing time decreased from **451 seconds to 402.67 seconds** with parallel execution, demonstrating a notable

Dataset	Execution Type	Average Landing Time
NOAA Weather Data	Non-Parallel	451±36.0056 sec
	2 Parallel	402.67±13.23 sec
CERT Insider Threat Data	Non-Parallel	861 sec
	2 Parallel	763 sec
Stock Market Data	Non-Parallel	2,643 sec
	2 Parallel	2,342 sec

Table 6.5: Comparison of Non-Parallel and 2 Parallel Landing Batch Process for All Datasets

performance improvement. Additionally, the standard deviation dropped significantly from **36.0056 seconds to 13.23 seconds**, indicating greater consistency in the landing times.

- **CERT Insider Threat Data:** Parallel execution reduced the landing time from **861 seconds to 763 seconds**, a reduction of nearly **11.4%**. This improvement, while significant, couldn't be accompanied by standard deviation analysis due to the limited number of iterations.
- **Stock Market Data:** The landing time for the stock dataset was reduced from **2,643 seconds to 2,342 seconds**, an efficiency gain of approximately **11.4%**, similar to the CERT dataset. Like the CERT data, standard deviation values were not available due to single-run execution.

These findings validate the effectiveness of the asynchronous and parallel architecture in improving the data loading process, particularly for larger datasets. The parallel execution consistently reduced landing times across all datasets and improved the consistency of the NOAA dataset, as seen in the reduced standard deviation. The architecture demonstrates promising scalability and enhanced efficiency, especially in high-volume data processing environments.

6.5 Staging Batch Process

The performance of the proposed asynchronous and parallel architecture for Hive was analyzed in the staging batch process across three different datasets: NOAA Weather Data, CERT Insider Threat Data, and Stock Market Data. Each dataset was processed with a focus on staging time, with the NOAA dataset being tested 10 times to gather detailed performance statistics, while the CERT and Stock datasets were tested with a single iteration due to their larger size.

- **NOAA Weather Data:** The data collected from 10 iterations of the staging batch process for the NOAA Weather Data are shown in Table 6.6. Key performance metrics for the NOAA dataset are as follows:
  - **Average Staging Time:** The average (mean) staging time across 10 iterations was calculated to be 578.7 seconds. This value represents the central tendency of the data insertion times.

Staging Time (sec)	Normal Distribution
547	0.007562859
549	0.008140645
552	0.009009748
555	0.009864609
572	0.013447348
575	0.013701245
578	0.013810101
613	0.006823502
620	0.004968462
626	0.003612533

Table 6.6: Staging Batch Process - Normal Distribution Values (NOAA Dataset)

– **Standard Deviation:** The standard deviation was 28.88 seconds, reflecting the consistency of the process. A smaller deviation indicates that the staging times were closely clustered around the mean.

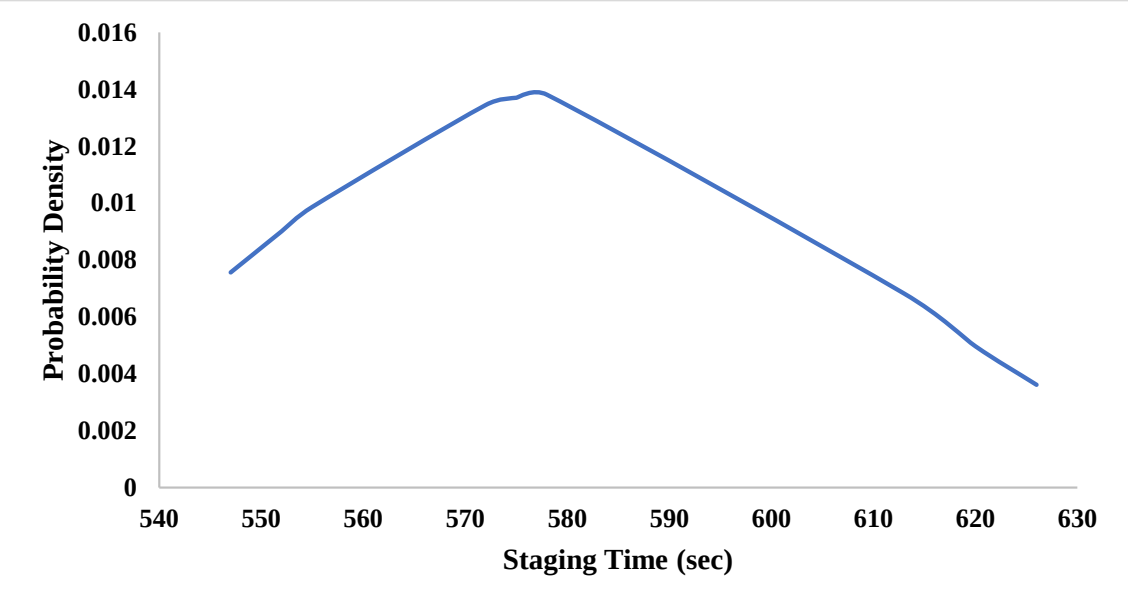

Figure 6.3: Normal Distribution of Staging Batch Process

– **Normal Distribution Analysis:** The normal distribution values provided for each staging time indicate how frequently each staging time occurs relative to the others. The Figure 6.3 of the normal distribution shows a bell-shaped curve, which is characteristic of a normal distribution. The peak of the curve corresponds to the average staging time, and the spread of the curve is influenced by the standard deviation.

– **Consistency of Staging Times:** The average staging time of 578.7 seconds (around 9.6 minutes) indicates that the process was stable, with most times falling within the range of 549.82 to 607.58 seconds (mean ± one standard deviation).

- **CERT Insider Threat Data:** The CERT Insider Threat dataset is considerably larger and more complex than the NOAA dataset. As a result, staging was tested once for this dataset. The key performance metric is:
  - **Staging Time:** The staging time for the CERT dataset was 1,121 seconds (around 18.7 minutes), significantly higher than the NOAA dataset due to the dataset's larger size and complexity.
- **Stock Market Data:** The Stock Market dataset, being the largest and most complex of the three, also had its staging batch process tested once. The performance metric is:
  - **Staging Time:** The staging time for the Stock dataset was 3,401 seconds (around 56.7 minutes), indicating the heavy load and complexity involved in processing this dataset.

6.6 Performance Comparison

This section compares the performance of the traditional architecture and the proposed asynchronous and parallel architecture across three datasets: NOAA Weather Data, CERT Insider Threat Data, and Stock Market Data. The comparison highlights improvements in both landing and total processing times, focusing on parallelism and its effects on performance.

6.6.1 Analysis

1. **Traditional Architecture:** The traditional architecture serves as the baseline and lacks any form of parallel processing. This results in the longest landing and total processing times across all datasets. For instance, in the NOAA dataset, the traditional landing time was 451 seconds, with a total time of 1,029.7 seconds. This setup did not require iterative landing processes. Similarly, the CERT dataset took 861 seconds for landing and 1,982 seconds in total, while the Stock dataset had a landing time of 2,643 seconds and a total time of 6,044 seconds.
2. **Proposed Architecture (No Parallelism):**
   - In the proposed architecture without parallel processing, while the landing time remained the same as the traditional architecture, the total processing time was significantly reduced.
     - For the NOAA dataset, the landing time stayed at 451 seconds, but the total time was reduced to 599.20 seconds, marking a 41.81% improvement over the traditional architecture.
     - In the CERT dataset, the landing time was also identical to the traditional method (861 seconds), but the total time decreased to 1,142 seconds, yielding an improvement of 42.38%.
     - Similarly, for the Stock dataset, the landing time remained unchanged at 2,643 seconds, but the total time dropped to 3,421.98 seconds, providing an improvement of 43.38%.

- These results show the effectiveness of the proposed architecture even without parallelism, particularly in reducing total time.

3. **Proposed Architecture (Parallelism with 2 Processes):**

   - Introducing parallelism in the landing process further reduced the landing and total times.
     - In the NOAA dataset, with two parallel processes, the landing time was reduced to 402.66 seconds, and the total time fell to 597 seconds, resulting in a 42.02% improvement over the traditional approach.
     - In the CERT dataset, the landing time decreased to 763 seconds, with a total time of 1,139.61 seconds, yielding a slightly better improvement of 42.50% over the traditional method.
     - In the Stock dataset, parallelism reduced the landing time to 2,342 seconds, and the total time to 3,419.59 seconds, resulting in an improvement of 43.42%.
   - These results highlight the efficiency of parallel processing in the proposed architecture, particularly in reducing landing times and further optimizing total processing times, especially for large datasets like CERT and Stock.

6.6.2 Key Observations:

- **Significant Time Reductions:** The proposed architecture significantly outperforms the traditional approach across all datasets, showing more than 40% reductions in total processing time. This improvement is consistent, regardless of whether parallelism is applied or not.

- **Parallelism Impact:** The use of two parallel processes consistently reduced landing times in all datasets, leading to further reductions in total processing time. However, the improvements in total time between non-parallel and parallel processing were modest, due to the unchanged staging times.

- **Dataset Size Sensitivity:** Larger datasets, such as CERT and Stock, benefited more from parallel processing in terms of both landing and total time reductions. The Stock dataset showed the greatest improvement in landing time with a 43.42% total time improvement.

6.6.3 Detailed Dataset-Specific Performance Comparison

6.6.3.1 NOAA Weather Data

The following Table 6.7 shows the performance comparison between the traditional architecture and the proposed architecture for the NOAA dataset, with and without parallelism:

Architecture	Parallelism in Landing Batch Process	Landing Time (sec)	Avg. Landing Iteration Time (sec)	Staging Time (sec)	Total Time (sec)	Improvement (%)
Traditional	Not Applicable	451.00	Not Required	578.70	1,029.70	-
Proposed	Landing with no parallel	451.00	20.50	578.70	599.20	41.81%
Proposed	Landing with parallel 2	402.66	18.30	578.70	597.00	42.02%

Table 6.7: Performance Comparison of Traditional and Proposed Architectures for NOAA Dataset

6.6.3.2 CERT Insider Threat Data

The CERT dataset results show similar trends as NOAA but with higher baseline times due to the larger dataset size in Table 6.8.

Architecture	Parallelism in Landing Batch Process	Landing Time (sec)	Avg. Landing Iteration Time (sec)	Staging Time (sec)	Total Time (sec)	Improvement (%)
Traditional	Not Applicable	861.00	Not Required	1,121.00	1,982.00	-
Proposed	Landing with no parallel	861.00	21.00	1,121.00	1,142.00	42.38%
Proposed	Landing with parallel 2	763.00	18.61	1,121.00	1,139.61	42.50%

Table 6.8: Performance Comparison of Traditional and Proposed Architectures for Threat Data

6.6.3.3 Stock Market Data

The Stock Market dataset, being the largest among the three, demonstrates the greatest total time improvements in Table 6.9.

Architecture	Parallelism in Landing Batch Process	Landing Time (sec)	Avg. Landing Iteration Time (sec)	Staging Time (sec)	Total Time (sec)	Improvement (%)
Traditional	Not Applicable	2,643.00	Not Required	3,401.00	6,044.00	-
Proposed	Landing with no parallel	2,643.00	20.98	3,401.00	3,421.98	43.38%
Proposed	Landing with parallel 2	2,342.00	18.59	3,401.00	3,419.59	43.42%

Table 6.9: Performance Comparison of Traditional and Proposed Architectures for Stock Data

6.6.4 Overall Improvement Comparison

The Table 6.10 summarizes the improvements in landing and total time across all datasets, comparing the improvements achieved by the proposed architecture with and without parallel processing:

Source	**Landing Improvement (%)**	**Total Improvement (%) Landing No Parallel**	**Total Improvement (%) Landing Parallel 2**
NOAA	10.72%	41.81%	42.02%
Threat	11.38%	42.38%	42.50%
Stock	11.39%	43.38%	43.42%

Table 6.10: Comparison of Landing and Total Improvements Across Different Sources

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis introduces an innovative asynchronous and parallel architecture for optimizing data loading in Hive. The architecture overcomes the limitations of traditional sequential processing, delivering significant performance improvements across multiple datasets, including NOAA, Threat, and Stock. By leveraging parallelism, the proposed design successfully accelerates data ingestion and reduces total processing time, marking a substantial advancement in big data management.

Key findings of the study include:

- **Significant Performance Gains:** The proposed architecture demonstrated consistent performance improvements across all three datasets. For NOAA, the total processing time was reduced by 42%, while the Threat dataset saw a 42.5% improvement, and the Stock dataset showed the highest improvement at 43.42%. These results affirm the scalability and effectiveness of the architecture in different data environments.

- **Enhanced Landing Efficiency:** Parallelism led to significant reductions in landing time. For NOAA data, landing time was reduced from 451 seconds to 402.66 seconds. Similarly, Threat data saw a reduction from 861 to 763 seconds, and Stock data from 2,643 to 2,342 seconds. This demonstrates the efficiency of parallelism in managing varying data sizes.

- **Iterative Process Optimization:** Parallel processing also optimized iteration times across the datasets. For NOAA, the average landing iteration time dropped from 20.50 to 18.30 seconds. In the Threat dataset, it reduced from 21.00 to 18.61 seconds, and in the Stock dataset, from 20.98 to 18.59 seconds. This improvement in iteration speed contributes to overall processing efficiency.

- **Potential for Further Optimization:** Despite these gains, staging times remained consistent across all datasets, suggesting that further optimizations could unlock even greater performance improvements, particularly for large-scale datasets like Stock.

The results unequivocally demonstrate that the proposed asynchronous and parallel data loading architecture outperforms conventional methods, establishing it as a superior alternative for modern data processing needs in Hive.

7.2 Limitations

Although the architecture presents significant advancements, there are some limitations:

- **Staging Process Performance:** The staging phase, despite improvements in landing times, did not see a parallelized performance boost. For all datasets, staging times remained static, suggesting a need for further investigation into parallelizing or optimizing this phase.
- **Fixed Parallelism Configuration:** This study explored the effects of using two parallel processes. However, the impact of different levels of parallelism (e.g., 4 or more processes) remains unexplored. It's unclear how the architecture would behave with different configurations, especially for larger datasets like the Stock data.
- **Resource Utilization and Efficiency:** The introduction of parallel processing likely increases the demand for computational resources (CPU, memory), but a comprehensive cost-benefit analysis was not performed. The trade-off between improved performance and resource utilization needs further exploration.
- **Data Skew and Load Balancing:** The architecture assumes even data distribution across parallel processes. However, in real-world scenarios, data skew could affect efficiency, potentially requiring advanced load balancing mechanisms for optimal performance.
- **Variable Factors - Network Congestion and Bandwidth:** The results of this experiment were obtained under idealized conditions, with variables like network congestion and bandwidth excluded from consideration. As such, while the findings demonstrate significant performance improvements in data loading, real-world implementations may encounter additional challenges related to these factors.
- **Limited Validation Scope:** The testing environment was confined to a Hadoop-based Cloudera setup. Broader validation across other platforms (e.g., cloud-based data lakes, different big data frameworks) and data formats would provide a more comprehensive understanding of the architecture's versatility.

7.3 Future Work

To build on these transformative findings, future research should focus on:

- **Advanced Staging Optimization:** Explore strategies to parallelize the staging process or optimize data transformations, aiming to further reduce processing time and maximize the benefits of the proposed architecture.

- **Scalability and Parallelism Exploration:** Investigate the impact of varying levels of parallelism (e.g., 4, 8, or more parallel processes) to determine the scalability limits and identify the optimal configuration for different workloads.
- **Hybrid Processing with Real-Time Integration:** Examine the potential of integrating the proposed architecture with real-time data processing frameworks like Apache Kafka or Flink, enabling a hybrid approach that combines batch and stream processing for more dynamic data environments.
- **Resource Efficiency and Cost-Benefit Analysis:** Conduct a detailed evaluation of resource consumption and cost implications to balance the performance improvements with the overhead introduced by parallel processing.
- **Implementation of Load Balancing Techniques:** Test advanced load balancing algorithms to enhance data distribution across parallel processes, ensuring even greater efficiency and performance stability.
- **Validation Across Diverse Platforms:** Extend the research to various data environments, including cloud data lakes, enterprise-scale data warehouses, and different data formats, to confirm the robustness and versatility of the proposed architecture.

Bibliography

[1] R. J. Santos and J. Bernardino, "Real-time data warehouse loading methodology," in *Proceedings of the 2008 international symposium on Database engineering & applications*, 2008, pp. 49–58.

[2] R. J. Santos and J. Bernardino, "Optimizing data warehouse loading procedures for enabling useful-time data warehousing," in *Proceedings of the 2009 International Database Engineering & Applications Symposium*, 2009, pp. 292–299.

[3] P. Vassiliadis and A. Simitsis, "Extraction, transformation, and loading.," *Encyclopedia of Database Systems*, vol. 10, 2009.

[4] A. Thusoo, J. S. Sarma, N. Jain, *et al.*, "Hive-a petabyte scale data warehouse using hadoop," in *2010 IEEE 26th international conference on data engineering (ICDE 2010)*, IEEE, 2010, pp. 996–1005.

[5] A. Abouzied, D. J. Abadi, and A. Silberschatz, "Invisible loading: Access-driven data transfer from raw files into database systems," in *Proceedings of the 16th International Conference on Extending Database Technology*, 2013, pp. 1–10.

[6] T. Liu, J. Liu, H. Liu, and W. Li, "A performance evaluation of hive for scientific data management," in *2013 IEEE International Conference on Big Data*, IEEE, 2013, pp. 39–46.

[7] S. Sagiroglu and D. Sinanc, "Big data: A review," in *2013 international conference on collaboration technologies and systems (CTS)*, IEEE, 2013, pp. 42–47.

[8] S. K. Bansal, "Towards a semantic extract-transform-load (etl) framework for big data integration," in *2014 IEEE International Congress on Big Data*, IEEE, 2014, pp. 522–529.

[9] Y. Cheng and F. Rusu, "Parallel in-situ data processing with speculative loading," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1287–1298.

[10] Y. Huai, A. Chauhan, A. Gates, *et al.*, "Major technical advancements in apache hive," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1235–1246.

[11] M. Mesiti and S. Valtolina, "Towards a user-friendly loading system for the analysis of big data in the internet of things," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, IEEE, 2014, pp. 312–317.

[12] K. Sridhar and M. Sakkeer, "Optimizing database load and extract for big data era," in *Database Systems for Advanced Applications: 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part II 19*, Springer, 2014, pp. 503–512.

[13] A. U. Abdullahi, R. Ahmad, and N. M. Zakaria, "Big data: Performance profiling of meteorological and oceanographic data on hive," in *2016 3rd international conference on computer and information sciences (ICCOINS)*, IEEE, 2016, pp. 203–208.

[14] S. Shaw, A. F. Vermeulen, A. Gupta, *et al.*, "Loading data into hive," *Practical Hive: A Guide to Hadoop's Data Warehouse System*, pp. 99–114, 2016.

[15] E. Costa, C. Costa, and M. Y. Santos, "Efficient big data modelling and organization for hadoop hive-based data warehouses," in *European, Mediterranean, and Middle Eastern Conference on Information Systems*, Springer, 2017, pp. 3–16.

[16] J. Camacho-Rodríguez, A. Chauhan, A. Gates, *et al.*, "Apache hive: From mapreduce to enterprise-grade big data warehousing," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1773–1786.

[17] M. Rodrigues, M. Y. Santos, and J. Bernardino, "Big data processing tools: An experimental performance evaluation," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 2, e1297, 2019.

[18] C.-C. Yang and G. Cong, "Accelerating data loading in deep neural network training," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, IEEE, 2019, pp. 235–245.

[19] J. C. Nwokeji and R. Matovu, "A systematic literature review on big data extraction, transformation and loading (etl)," in *Intelligent Computing: Proceedings of the 2021 Computing Conference, Volume 2*, Springer, 2021, pp. 308–324.

[20] A. Adamov, "Large-scale data modelling in hive and distributed query processing using mapreduce and tez," *arXiv preprint arXiv:2301.12454*, 2023.